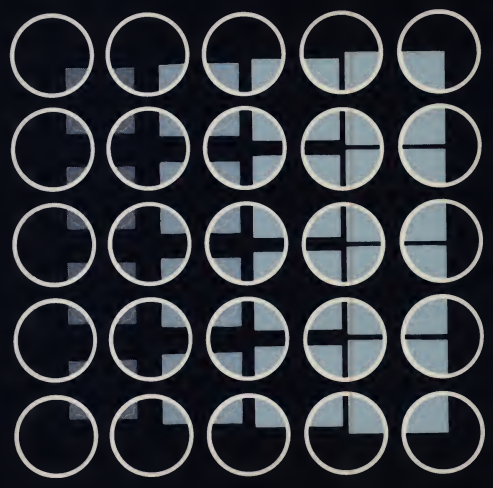
ULTRIX-32m™

Supplementary Documents, Vol. III
System Management



digital software

ULTRIX-32m™

Supplementary Documents

Volume 3

System Management

Order No. AA-CN32A-TE

digital equipment corporation, merrimack, new hampshire

Copyright © 1984 by Digital Equipment Corporation

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such a license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	Edusystem	ULTRIX
DEC/CMS	IAS	ULTRIX-11
DEC/MMS	MASSBUS	ULTRIX-32
DECnet	MICRO/PDP-11	ULTRIX-32m
DECsystem-10	Micro/RSX	UNIBUS
DECsystem-20	PDP	VAX
DECUS	PDT	VMS
DECwriter	RSTS	VT
DIBOL	RSX	digital

UNIX is a trademark of AT&T Bell Laboratories.

Information herein is derived from copyrighted material as permitted under a license agreement with AT&T Bell Laboratories.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the Electrical Engineering and Computer Sciences Departments at the Berkeley Campus of the University of California for their role in its development.

The following command descriptions as set forth in this document are copyrighted material of Digital Equipment Corporation:

mdtar(1)
adduser(1)
addgroup(1)
netsetup(1)
uucpsetup(1)
removeuser(1)

In accordance with the licenses granted to Digital Equipment Corporation by AT&T Bell Laboratories and the University of California at Berkeley pertaining to the software described herein, the following should be understood by the licensee, and any related documentation provided by the licensee to third parties, whether pursuant to an agreement with Digital Equipment Corporation permitting sublicensing of the software described herein or otherwise, must contain the provisions of this section as set forth below:

Information herein is derived from copyrighted material as permitted under a license agreement with AT&T Bell Laboratories.

Copyright © 1979 AT&T Bell Laboratories.

UNIX is a trademark of AT&T Bell Laboratories. The UNIX trademark may not be used in the name of any licensee's product. Any use of the trademark in advertising, publicity, packaging, labeling or otherwise must state that UNIX is a trademark of AT&T Bell Laboratories.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development:

The Electrical Engineering and Computer Science Department at the Berkeley Campus of the University of California. Eric Allman, Ken Arnold, Ozalp Babaoglu, Scott B. Baden, Jerry Berkman, John Breedlove, Earl T. Cohen, Robert P. Corbett, Mike Curry, Steve Feldman, Tom Ferrin, John Foderaro, Susan L. Graham, Charles Haley, Robert R. Henry, Andy Hertzfeld, Mark Horton, S.C. Johnson, William Joy, Howard Katseff, Peter Kessler, Jim Kleckner, J. E. Kulp, James Larus, Kevin Layer, Mike Lesk, Steve Levine, Jeff Levinsky, Louise Madrid, M. Kirk McKusick, Colin L. McMaster, Mikey Olson, Geoffrey Peck, Ed Pelegri-Llopart, Rob Pike, Dave Presotto, John F. Reiser, Asa Romberger, Bill Rowan, Jeff Schreibman, Eric P. Scott, Greg Shenaut, Eric Shienbrood, Kurt Shoens, Keith Sklower, Helge Skrivervik, Al Stanberger, Ken Thompson, Michael C. Toy, Richard Tuck, Bill Tuthill, Mike Urban, Edward Wang, David Wasley, Joseph Weizenbaum, Jon L. White, Glenn Wichman, and Niklaus Wirth.

"The UNIX Time-Sharing System": Copyright © 1974, Association for Computing Machinery, Inc. reprinted by permission. This is a revised version of an article that appeared in Communications of the ACM, 17, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973. Acknowledgements: for their help and support, R.H. Canaday, R. Morris, M.D. McIlroy, and J.F. Ossanna.

"Advanced Editing on UNIX" acknowledgement: Ted Dolotta for his ideas and assistance.

"Ex Reference Manual" acknowledgements: Chuck Haley contributed greatly to the early development of ex. Bruce Englar encouraged the redesign which led to ex version 1. Bill Joy wrote versions 1 and 2.0 through 2.7, and created the framework that users see in the present editor. Mark Horton added macros and other features and made the editor work on a large number of terminals and UNIX systems.

"An Introduction to the UNIX Shell" acknowledgements: Dennis Ritchie, John Mashey and Joe Maranzano for their help and support.

"An Introduction to the C Shell" acknowledgements: Michael Ubell, Eric Allman, Mike O'Brien, and Jim Kulp.

"A System for Typesetting Mathematics" acknowledgements: J.F. Ossanna, A.V. Aho, and S.C. Johnson, for their ideas and assistance.

"A TROFF Tutorial" acknowledgements: J. F. Ossanna, Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman, for their help and support.

The document "The C Programming Language - Reference Manual" is reprinted, with minor changes, from "The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

"Make - A Program for Maintaining Computer Programs" acknowledgements: S.C. Johnson, and H. Gajewska, for their ideas and assistance.

"Screen Updating and Cursor Movement Optimization: A Library Package" acknowledgements: For their help and support, Bill Joy, Doug Merritt, Kurt Shoens, Ken Abrams, Alan Char, Mark Horton, and Joe Kalash.

"YACC: Yet Another Compiler-Compiler" acknowledgements: B.W. Kernighan, P.J. Plauger, S.I. Feldman, C. Imagna, M.E. Lesk, A. Snyder, C.B. Haley, D.M. Ritchie, M.O. Harris and Al Aho, for their ideas and assistance.

"Lex - A Lexical Analyzer Generator" acknowledgements: S.C. Johnson, A.V. Aho, and Eric Schmidt, for their help as originators of much of Lex, as well as debuggers of it.

"The M4 Macro Processor" acknowledgements: Rick Becker, John Chambers, Doug McIlroy, and Jim Weythman, for the help and support.

"BC - An Arbitrary Precision Desk-Calculator Language" acknowledgement: The compiler is written in YACC; its original version was written by S.C. Johnson.

"A Dial-Up Network of UNIX TM Systems" acknowledgements: G.L. Chesson, A.S. Cohen, J. Lions, and P.F. Long, for their suggestions and assistance.

The Fourth Berkeley Software Distribution is provided by the Regents of the University of California and the Other Contributors on an "as is" basis. Neither the Regents of the University of California nor the Other Contributors warrant that the functions contained in the Fourth Berkeley Software Distribution will meet the licensee's requirements or will operate in the combinations which may be selected for use by the licensee, or that the operation of the Fourth Berkeley Software Distribution will be uninterrupted or error free. Neither the Regents of the University of California nor the Other Contributors make any warranties, either express or implied, as to any matter whatsoever, including without limitation, the condition of the Fourth Berkeley Software Distribution, its merchantability or its fitness for any particular purpose.

The licensee understands and agrees that the Regents of the University of California and the Other Contributors are under no obligation to provide either maintenance services, update services, notices of latent defects, or corrections of defects for Fourth Berkeley Software Distribution.

CONTENTS

BEFORE YOU START

PART 1: OPERATING SYSTEM CHANGES

A FAST FILE SYSTEM FOR UNIX 1

Introduction
Old File System
New File System Organization
Performance
File System Functional Enhancements
Software Engineering

PART 2: MAINTENANCE AND ADMINISTRATION

DISC QUOTAS IN A UNIX ENVIRONMENT 20

Users' View of Disc Quotas
Administering the Quota System
Some Implementation Detail

FSCK — THE UNIX FILE SYSTEM CHECK PROGRAM 24

Introduction
Overview of the File System
Fixing Corrupted File Systems
Appendix A: Fsck Error Conditions

SENDMAIL INSTALLATION AND OPERATION GUIDE 44

Basic Installation
Normal Operations
Arguments
Tuning
The Whole Scoop on the Configuration File
Appendix A: Command Line Flags
Appendix B: Configuration Options
Appendix C: Mailer Flags
Appendix D: Other Configuration
Appendix E: Summary of Support Files

PART 3: COMMUNICATIONS

A 4.2BSD INTERPROCESS COMMUNICATION PRIMER 81

Introduction
Basics
Network Library Routines
Client/Server Model
Advanced Topics

4.2BSD NETWORKING IMPLEMENTATION NOTES 107

Introduction
Overview
Goals
Internal Address Representation
Memory Management
Internal Layering
Socket/Protocol Interface
Protocol/Protocol Interface
Protocol/Network-Interface Interface
Gateways and Routing Issues
Raw Sockets
Buffering and Congestion Control
Out of Band Data
Trailer Protocols

SENDMAIL: AN INTERNETWORK MAIL ROUTER 136

Design Goals
Overview
Usage and Implementation
Comparison with Other Mailers
Evaluations and Future Plans

PART 4: SECURITY CONSIDERATIONS

ON THE SECURITY OF UNIX 151

PASSWORD SECURITY: A CASE HISTORY 154

Introduction
Prologue
The First Scheme
Attacks on the First Approach
An Anecdote
Improvements to the First Approach
Conclusions

PART 5: SUPPORTING DOCUMENTS

A DIAL–UP NETWORK OF UNIX SYSTEMS 161

Purpose
Design Goals
Processing
Present Uses
Performance
Further Goals
Lessons

UUCP IMPLEMENTATION DESCRIPTION 169

UUCP — UNIX to UNIX File Copy
UUX — UNIX to UNIX Execution
UUCICO — Copy In, Copy Out
UUXQT — UUCP Command Execution
UULOG — UUCP Log Inquiry
UUCLEAN — UUCP Spool Directory Cleanup
Security
UUCP Installation
Administration

UUCP INSTALLATION AND ADMINISTRATION 183

Introduction
Installation
UUCP Administration and Maintenance
APPENDIX – Summary of Enhancements and Quick Installation Guide

ULTRIX–32m Supplementary Documents
Volume III System Managers

Order No. AA–CN32A–TE

digital equipment corporation * merrimack, new hampshire

BEFORE YOU START

This is the third volume of *ULTRIX–32m Supplementary Documents*, a three volume set that contains articles describing the ULTRIX–32m system. The authors are computer scientists and program developers at Bell Laboratories and the University of California at Berkeley. The articles explain the software tools and utilities available on your ULTRIX–32m system. They constitute most of the lore that enriches this operating system; topics range from getting started to the details of screen updating and cursor movement facilities.

Each volume in this set contains several parts, and each part begins with an introduction. Each introduction serves as a map that will help you find your way around in the documentation, allowing you to select articles that relate to your interest. Each introduction gives an overview of the material covered in the part and a description of the articles included. Most readers will not need to read all articles in any part, since many articles cover parallel topics.

These articles provide authoritative and accurate information that is unavailable elsewhere. However, you should be aware that some of the information in some articles is dated. We include those articles because many of the concepts they develop are still current and important.

At the end of each volume in this set, you will find a master index identifying topics and related pages in the text for all three volumes.

Topics in Volume III

The articles in this third volume are written for people responsible for the installation, administration, and daily maintenance of the ULTRIX–32m system.

"A Fast File System for UNIX," by McKusic, Joy, Leffler, and Fabry, compares the new file system used in ULTRIX–32m with the original UNIX file system. The new system is faster and more reliable, and the block size is adjustable. The article also explains considerations and procedures that will help you take full advantage of these improvements.

The articles in Part 2, Maintenance and Administration, deal with disk quotas, fixing corrupted file systems, and management of the *sendmail* program. The *quota* utility enables the system manager to limit the number of blocks and the number of files available to each user. *Fsck*, the File System Check Program, lets you examine the integrity of the file system and repair any inconsistencies. The *sendmail* program lets users send messages between computer systems that are connected to different networks.

Articles in Part 3, Communications, explain the interprocess communication software. Articles in Part 4, Security Considerations, offer a variety of tips on how you can protect your system against crashes and unauthorized access. And Part 5, Supporting Documents, provides information on software changes new to this release.

PART 1: OPERATING SYSTEM CHANGES

"A Fast File System for UNIX," by McKusick, Joy, Leffler, and Fabry, explains the new file system in detail. The article is essential to people responsible for management and administration of ULTRIX–32m systems.

The new file system, unlike the original UNIX file system, allows you to select a block size. The block size can be 4096 bytes or 8192 bytes; you must choose the size when you create the file system. You can optimize the disk usage and file transfer rates on your ULTRIX–32m system by choosing a block size that:

- Matches the physical characteristics of your disk drives
- Is appropriate for your applications

The article also explains use of:

- A file-locking facility that allows cooperating programs to apply advisory locks on files
- Symbolic links that allow references across separate physical file systems
- A rename facility that replaces three system calls with one
- A *quota* utility that allows the system administrator to set limits on the number of blocks and the number of files available to each user

You can find more detailed information on the *quota* utility in "Disk Quotas in a UNIX Environment" in Part 2 of this volume.

A Fast File System for UNIX*

Revised July 27, 1983

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies, that allow better locality of reference and that can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access for large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the user interface are discussed. These include a mechanism to lock files, extensions of the name space across file systems, the ability to use arbitrary length file names, and provisions for efficient administrative control of resource usage.

---

* UNIX is a trademark of Bell Laboratories.

†William N. Joy is currently employed by: Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043

‡Samuel J. Leffler is currently employed by: Lucasfilm Ltd., PO Box 2009, San Rafael, CA 94912

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system*.

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

---

† A file system always resides on a single drive.

* The actual number may vary from system to system, but is usually in the range 5-13.

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2↑32 bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space. The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

Table 1 - Amount of wasted space as a function of block size.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 - Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0-5, 10, and 11 are in use, while fragments 6-9, and 12-15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6-9 cannot be coalesced into a block; only fragments 12-15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.
2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.
3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.
4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to insure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Type of File System	Processor and Bus Measured	Read Speed	Read Bandwidth	Read % CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 2a - Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write Speed	Write Bandwidth	Write % CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Table 2b - Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

* This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the

comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

---

system's PDP-11 ancestry.

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time, by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a protection scheme that was contrary to the UNIX philosophy or could not be used by system

administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a *raw disk*. This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code (including comments).

Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when files were less stable than they should have been. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

References

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Porcar82] Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[Thompson79] Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

PART 2: MAINTENANCE AND ADMINISTRATION

The three articles in this part describe system administration utilities on ULTRIX–32m. Two of the utilities, *quota* and the file system check program (*fsck*), will help you keep your system running efficiently. The third utility, *sendmail*, makes possible communication between users on computers that use different networking software.

Disk Quotas

The ULTRIX–32m system allows the system manager to impose limits on the amount of disk space and the number of files available to each user. Each category (disk space and the maximum number of files) has a hard limit and a soft limit. The hard limit for a user sets an absolute maximum that cannot be exceeded. The soft limit is a guideline: the number of blocks or files that the user should try not to exceed. The *quota* utility warns any user who exceeds his or her soft limit. If the user consistently ignores the warnings, the soft limit becomes a hard limit after a set number of warnings.

The article, "Disc Quotas in a UNIX Environment," by Elz, tells how the system manager can establish, disable, or check the limits and the number of warnings for any user. Elz also explains how a user can exit without loss from an editing session in which writing the edited material to a file would exceed one of the hard limits.

Fixing Corrupted File Systems

The ULTRIX–32m system includes a file system check program called *fsck*. You can use this utility to determine whether your file system is corrupted and to fix any inconsistencies you find.

Fsck runs in two modes: noninteractive and interactive. Normally the boot procedure calls *fsck* to run noninteractively after booting the operating system. In this mode, the utility checks for inconsistencies and corrects only those that it can handle without help from an operator. In general, these are problems associated with a system crash or improper shutdown procedure. When the utility finds a problem it can't deal with, it notifies the operator and stops. The operator can then run *fsck* interactively, deciding between the alternative measures presented by the utility.

The article by McKusick, "Fsck — The UNIX File System Check Program," gives an overview of the file system, the kinds of corruption that can occur, and the methods that *fsck* uses to check for inconsistencies. An appendix provides a comprehensive list of error messages together with explanations and appropriate responses. *Fsck* is essential to proper maintenance of the ULTRIX–32m system, and this article is essential to proper use of *fsck*.

Managing the Sendmail Utility

Sendmail is an internetwork mail utility transparent to most users. Once it is installed and running, you can send mail to users on foreign network systems in the same way that you send mail to users on the local network. The *sendmail* utility handles the protocol and message-routing differences between networks automatically.

The "Sendmail Installation and Operation Guide," by Allman, tells what you need to know to start up the utility and to keep it running correctly from day to day. A second article, "Sendmail — An Internetwork Mail Router," in Part 3 of this volume, gives background information that tells how *sendmail* works. Read the background article before using the installation and operating information included in this part.

The installation information in the "Sendmail Installation and Operation Guide" explains:

- How to use either of two off-the-shelf configuration files supplied with the software
- How to use a makefile to install *sendmail* automatically
- How to install *sendmail* by hand by building your own configuration file and setting up the *sendmail* startup procedure on your ULTRIX–32m system

The day-to-day *sendmail* operations explained include:

- Use of the system log for records and debugging
- Mail queue processing
- Treatment of address aliases
- The mail-forwarding feature
- Special headers for return receipts and error situations

The article describes parameters you can adjust to tune *sendmail* to suit a specific site. If you must build your own configuration file, you will find the list of configuration file rules and hints to be helpful. And for expert system managers, the appendixes list detailed *sendmail* information in five categories:

- Command line flags
- Configuration options
- Mailer flags
- Compilation options (other configuration)
- Support files

Disc Quotas in a UNIX* Environment.

Robert Elz

Department of Computer Science
University of Melbourne,
Parkville,
Victoria,
Australia.

1. Users' view of disc quotas

To most users, disc quotas will either be of no concern, or a fact of life that cannot be avoided. The *quota*(1) command will provide information on any disc quotas that may have been imposed upon a user.

There are two individual possible quotas that may be imposed, usually if one is, both will be. A limit can be set on the amount of space a user can occupy, and there may be a limit on the number of files (inodes) he can own.

Quota provides information on the quotas that have been set by the system administrators, in each of these areas, and current usage.

There are four numbers for each limit, the current usage, soft limit (quota), hard limit, and number of remaining login warnings. The soft limit is the number of 1K blocks (or files) that the user is expected to remain below. Each time the user's usage goes past this limit, he will be warned. The hard limit cannot be exceeded. If a user's usage reaches this number, further requests for space (or attempts to create a file) will fail with an EDQUOT error, and the first time this occurs, a message will be written to the user's terminal. Only one message will be output, until space occupied is reduced below the limit, and reaches it again, in order to avoid continual noise from those programs that ignore write errors.

Whenever a user logs in with a usage greater than his soft limit, he will be warned, and his login warning count decremented. When he logs in under quota, the counter is reset to its maximum value (which is a system configuration parameter, that is typically 3). If the warning count should ever reach zero (caused by three successive logins over quota), the particular limit that has been exceeded will be treated as if the hard limit has been reached, and no more resources will be allocated to the user. The **only** way to reset this condition is to reduce usage below quota, then log in again.

1.1. Surviving when quota limit is reached

In most cases, the only way to recover from over quota conditions, is to abort whatever activity was in progress on the filesystem that has reached its limit, remove sufficient files to bring the limit back below quota, and retry the failed program.

However, if you are in the editor and a write fails because of an over quota situation, that is not a suitable course of action, as it is most likely that initially attempting to write the file will have truncated its previous contents, so should the editor be aborted without correctly writing the file not only will the recent changes be lost, but possibly much, or even all, of the data that previously existed.

There are several possible safe exits for a user caught in this situation. He may use the editor ! shell escape command to examine his file space, and remove surplus files.

* UNIX is a trademark of Bell Laboratories.

Alternatively, using *csh*, he may suspend the editor, remove some files, then resume it. A third possibility, is to write the file to some other filesystem (perhaps to a file on /tmp) where the user's quota has not been exceeded. Then after rectifying the quota situation, the file can be moved back to the filesystem it belongs on.

2. Administering the quota system

To set up and establish the disc quota system, there are several steps necessary to be performed by the system administrator.

First, the system must be configured to include the disc quota sub-system. This is done by including the line:

options QUOTA

in the system configuration file, then running *config*(8) followed by a system configuration*.

Second, a decision as to what filesystems need to have quotas applied needs to be made. Usually, only filesystems that house users' home directories, or other user files, will need to be subjected to the quota system, though it may also prove useful to also include **/usr**. If possible, **/tmp** should usually be free of quotas.

Having decided on which filesystems quotas need to be set upon, the administrator should then allocate the available space amongst the competing needs. How this should be done is (way) beyond the scope of this document.

Then, the *edquota*(8) command can be used to actually set the limits desired upon each user. Where a number of users are to be given the same quotas (a common occurrence) the **–p** switch to edquota will allow this to be easily accomplished.

Once the quotas are set, ready to operate, the system must be informed to enforce quotas on the desired filesystems. This is accomplished with the *quotaon*(8) command. *Quotaon* will either enable quotas for a particular filesystem, or with the **–a** switch, will enable quotas for each filesystem indicated in **/etc/fstab** as using quotas. See *fstab*(5) for details. Most sites using the quota system, will include the line

/etc/quotaon -a

in **/etc/rc.local**.

Should quotas need to be disabled, the *quotaoff*(8) command will do that, however, should the filesystem be about to be dismounted, the *umount*(8) command will disable quotas immediately before the filesystem is unmounted. This is actually an effect of the *umount*(2) system call, and it guarantees that the quota system will not be disabled if the umount would fail because the filesystem is not idle.

Periodically (certainly after each reboot, and when quotas are first enabled for a filesystem), the records retained in the quota file should be checked for consistency with the actual number of blocks and files allocated to the user. The *quotachk*(8) command can be used to accomplish this. It is not necessary to dismount the filesystem, or disable the quota system to run this command, though on active filesystems inaccurate results may occur. This does no real harm in most cases, another run of *quotachk* when the filesystem is idle will certainly correct any inaccuracy.

The super-user may use the *quota*(1) command to examine the usage and quotas of any user, and the *repquota*(8) command may be used to check the usages and limits for all users on a filesystem.

---

* See also the document "Building 4.2BSD UNIX Systems with Config".

3. Some implementation detail.

Disc quota usage and information is stored in a file on the filesystem that the quotas are to be applied to. Conventionally, this file is **quotas** in the root of the filesystem. While this name is not known to the system in any way, several of the user level utilities "know" it, and choosing any other name would not be wise.

The data in the file comprises an array of structures, indexed by uid, one structure for each user on the system (whether the user has a quota on this filesystem or not). If the uid space is sparse, then the file may have holes in it, which would be lost by copying, so it is best to avoid this.

The system is informed of the existence of the quota file by the *setquota*(2) system call. It then reads the quota entries for each user currently active, then for any files open owned by users who are not currently active. Each subsequent open of a file on the filesystem, will be accompanied by a pairing with its quota information. In most cases this information will be retained in core, either because the user who owns the file is running some process, because other files are open owned by the same user, or because some file (perhaps this one) was recently accessed. In memory, the quota information is kept hashed by user-id and filesystem, and retained in an LRU chain so recently released data can be easily reclaimed. Information about those users whose last process has recently terminated is also retained in this way.

Each time a block is accessed or released, and each time an inode is allocated or freed, the quota system gets told about it, and in the case of allocations, gets the opportunity to object.

Measurements have shown that the quota code uses a very small percentage of the system cpu time consumed in writing a new block to disc.

4. Acknowledgments

The current disc quota system is loosely based upon a very early scheme implemented at the University of New South Wales, and Sydney University in the mid 70's. That system implemented a single combined limit for both files and blocks on all filesystems.

A later system was implemented at the University of Melbourne by the author, but was not kept highly accurately, eg: chown's (etc) did not affect quotas, nor did i/o to a file other than one owned by the instigator.

The current system has been running (with only minor modifications) since January 82 at Melbourne. It is actually just a small part of a much broader resource control scheme, which is capable of controlling almost anything that is usually uncontrolled in unix. The rest of this is, as yet, still in a state where it is far too subject to change to be considered for distribution.

For the 4.2BSD release, much work has been done to clean up and sanely incorporate the quota code by Sam Leffler and Kirk McKusick at The University of California at Berkeley.

Fsck – The UNIX† File System Check Program

Revised July 28, 1983

Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document reflects the use of *fsck* with the 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

File System Check Program (*fsck*) is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. Unless there has been a hardware failure, *fsck* is able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fsck*. Both the program and the interaction between the program and the operator are described.

---

†UNIX is a trademark of Bell Laboratories.

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

1. Introduction

This document reflects the use of *fsck* with the 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. *Fsck* runs in two modes. Normally it is run non-interactively by the system after a normal boot. When running in this mode, it will only make changes to the file system that are known to always be correct. If an unexpected inconsistency is found *fsck* will exit with a non-zero exit status, leaving the system running single-user. Typically the operator then runs *fsck* interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether or not the suggested correction should be made.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. Overview of the file system

The file system is discussed in detail in [Mckusick83]; this section gives a brief overview.

2.1. Superblock

A file system is described by its *super-block*. The super-block is built when the file system is created (*newfs*(8)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, *newfs* replicates it to protect against catastrophic loss. The *default super block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the inode structure itself†. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096 byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks.

In order to create files with up to 2↑32 bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when *newfs* creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

2.2. Summary information

Associated with the super block is non replicated *summary information*. The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, inodes and directories in the file system.

2.3. Cylinder groups

The file system partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group includes inode slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of inodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one inode for each 2048 bytes of disk space; this is expected to be far more inodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the *i+1*st cylinder group is about one track further from the beginning of the cylinder group than it was for the *i*th cylinder group. In this way, the redundant information spirals down into the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks.

†The actual number may vary from system to system, but is usually in the range 5-13.

Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

2.4. Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000 byte file stored on a 4096/1024 byte file system. This file uses two full size blocks and a 3072 byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file, as needed.

2.5. Updates to the file system

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program does an operation to change the file system, such as a *write*, the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the inode information reflecting the change, is eventually written out to disk. The real disk write may not happen until long after the *write* system call has returned. Thus at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when a *sync*(2) is done (at 30 second intervals) by */etc/update*(8), or by manual operator intervention with the *sync*(8) command. If the system is halted without writing out the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two inodes. Such an inconsistency can occur when the system is halted before the pointer to the block in the old inode has been cleared in the copy of the old inode on the disk, and after the pointer to the block in the new inode has been written out to the copy of the new inode on the disk. Here, there is no deterministic method for deciding which inode should really claim the block. A similar problem can arise with a multiply claimed inode.

The problem with asynchronous inode updates can be avoided by doing all inode deallocations synchronously. Consequently, inodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly inodes are kept consistent by synchronously deleting, adding, or changing directory entries.

3. Fixing corrupted file systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or a mounted file system is taken off-line. The most common operator procedural failure is forgetting to *sync* the system before halting the CPU.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

3.1. Detecting and correcting corruption

Normally *fsck* is run non-interactively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this paper we assume that *fsck* is being run interactively, and all possible errors can be encountered. When an inconsistency is discovered in this mode, *fsck* reports the inconsistency for the operator to chose a corrective action.

A quiescent‡ file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system, or computed from other known values. The file system **must** be in a quiescent state when *fsck* is run, since *fsck* is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the inodes, the indirect blocks, and the data blocks containing directory entries.

3.2. Super-block checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or inodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of inodes, free-block count, and the free-inode count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The file-system size and layout information are the most critical pieces of information for *fsck*. While there is no way to actually check these sizes, since they are statically determined by *newfs*, *fsck* can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If *fsck* detects corruption in the static parameters of the default super-block, *fsck* requests the operator to specify the location of an alternate super-block.

3.3. Free block checking

Fsck checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, *fsck* checks that the number of free blocks plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

‡ I.e., unmounted and not being written on.

If anything is wrong with the block allocation maps, *fsck* will rebuild them, based on the list it has computed of allocated blocks.

The summary information associated with the super-block counts the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-inode count.

3.4. Checking the inode state

An individual inode is not as likely to be corrupted as the allocation information. However, because of the great number of active inodes, a few of the inodes are usually corrupted.

The list of inodes in the file system is checked sequentially starting with inode 2 (inode 0 marks unused inodes; inode 1 is saved for future generations) and progressing through the last inode in the file system. The state of each inode is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes must be one of six types: regular inode, directory inode, symbolic link inode, special block inode, special character inode, or socket inode. Inodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formated inode. An inode can get in this state if bad data is written into the inode list. The only possible corrective action is for *fsck* is to clear the inode.

3.5. Inode links

Each inode counts the total number of directory entries linked to the inode. *Fsck* verifies the link count of each inode by starting at the root of the file system, and descending through the directory structure. The actual link count for each inode is calculated during the descent.

If the stored link count is non-zero and the actual link count is zero, then no directory entry appears for the inode. If this happens, *fsck* will place the disconnected file in the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated. If this happens, *fsck* replaces the incorrect stored link count by the actual link count.

Each inode contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the inode. Since indirect blocks are owned by an inode, inconsistencies in indirect blocks directly affect the inode that owns it.

Fsck compares each block number claimed by an inode against a list of already allocated blocks. If another inode already claims a block number, then the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, *fsck* will perform a partial second pass over the inode list to find the inode of the duplicated block. The second pass is needed, since without examining the files associated with these inodes for correct content, not enough information is available to determine which inode is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), then the inode with the earliest modify time is usually incorrect, and should be cleared. If this happens, *fsck* prompts the operator to clear both inodes. The operator must decide which one should be kept and which one should be cleared.

Fsck checks the range of each block number claimed by an inode. If the block number is lower than the first data block in the file system, or greater than the last data block, then the block number is a *bad block number*. Many bad blocks in an inode are usually caused by an

indirect block that was not written to the file system, a condition which can only occur if there has been a hardware failure. If an inode contains bad block numbers, *fsck* prompts the operator to clear it.

3.6. Inode data size

Each inode contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. *Fsck* computes the actual number of data blocks and compares that block count against the actual number of blocks the inode claims. If an inode contains an incorrect count *fsck* prompts the operator to fix it.

Each inode contains a thirty-two bit size field. The size is the number of data bytes in the file associated with the inode. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the inode, and comparing that expected block count against the actual number of blocks the inode claims.

3.7. Checking the data associated with an inode

An inode can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are: plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks contain the information stored in a file; symbolic link data blocks contain the path name stored in a link. Directory data blocks contain directory entries. *Fsck* can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory inode numbers pointing to unallocated inodes, directory inode numbers that are greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories that are not attached to the file system. If the inode number in a directory data block references an unallocated inode, then *fsck* will remove that directory entry. Again, this condition can only arise when there has been a hardware failure.

If a directory entry inode number references outside the inode list, then *fsck* will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." must be the first entry in the directory data block. The inode number for "." must reference itself; e.g., it must equal the inode number for the directory data block. The directory inode number entry for ".." must be the second entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, *fsck* will replace them with the correct values.

3.8. File system connectivity

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, then *fsck* links the directory back into the file system in the *lost+found* directory. This condition only occurs when there has been a hardware failure.

Acknowledgements

I thank Bill Joy, Sam Leffler, Robert Elz and Dennis Ritchie for their suggestions and help in implementing the new file system. Thanks also to Robert Henry for his editorial input to get this document together. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. (Kirk McKusick, July 1983)

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS. (T. Kowalski, July 1979)

References

[Dolotta78] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1* (January 1978).

[Joy83] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. *4.2BSD System Manual*, University of California at Berkeley, Computer Systems Research Group Technical Report #4, 1982.

[McKusick83] McKusick, M., Joy, W., Leffler, S., and Fabry, R. *A Fast File System for UNIX*, University of California at Berkeley, Computer Systems Research Group Technical Report #7, 1982.

[Ritchie78] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1905-29.

[Thompson78] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1931-46.

4. Appendix A – Fsck Error Conditions

4.1. Conventions

Fsck is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the map of free blocks, (possibly rebuilding it), and performs some cleanup.

Normally *fsck* is run non-interactively to *preen* the file systems after an unclean halt. While preen'ing a file system, it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this appendix many errors have several options that the operator can take. When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. When preen'ing most errors are fatal. For those that are expected, the response taken is noted. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

4.2. Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file. All of the initialization errors are fatal when the file system is being preen'ed.

C option?
C is not a legal option to *fsck*; legal options are -b, -y, -n, and -p. *Fsck* terminates on this error condition. See the *fsck*(8) manual entry for further detail.

cannot alloc NNN bytes for blockmap
cannot alloc NNN bytes for freemap
cannot alloc NNN bytes for statemap
cannot alloc NNN bytes for lncntp
Fsck's request for memory for its virtual memory tables failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't open checklist file: *F*
The file system checklist file *F* (usually */etc/fstab*) can not be opened for reading. *Fsck* terminates on this error condition. Check access modes of *F*.

Can't stat root
Fsck's request for statistics about the root directory "/" failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't stat *F*
Can't make sense out of name *F*
Fsck's request for statistics about the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

Can't open *F*
Fsck's request attempt to open the file system *F* failed. When running manually, it ignores this

file system and continues checking the next file system given. Check access modes of *F*.

F: **(NO WRITE)**
Either the -n flag was specified or *fsck*'s attempt to open the file system *F* for writing failed. When running manually, all the diagnostics are printed out, but no modifications are attempted to fix them.

file is not a block or character device; OK
You have given *fsck* a regular file name by mistake. Check the type of the file specified.

Possible responses to the OK prompt are:

YES Ignore this error condition.

NO ignore this file system and continues checking the next file system given.

One of the following messages will appear:
MAGIC NUMBER WRONG
NCG OUT OF RANGE
CPG OUT OF RANGE
NCYL DOES NOT JIVE WITH NCG*CPG
SIZE PREPOSTEROUSLY LARGE
TRASHED VALUES IN SUPER BLOCK

and will be followed by the message:
F: **BAD SUPER BLOCK:** *B*
USE -b OPTION TO FSCK TO SPECIFY LOCATION OF AN ALTERNATE SUPER-BLOCK TO SUPPLY NEEDED INFORMATION; SEE fsck(8).
The super block has been corrupted. An alternative super block must be selected from among those listed by *newfs* (8) when the file system was created. For file systems with a blocksize less than 32K, specifying -b 32 is a good first choice.

INTERNAL INCONSISTENCY: *M*
Fsck's has had an internal panic, whose message is specified as *M*. This should never happen. See a guru.

CAN NOT SEEK: BLK *B* **(CONTINUE)**
Fsck's request for moving to a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT READ: BLK *B* **(CONTINUE)**
Fsck's request for reading a specified block number *B* in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT WRITE: BLK *B* (CONTINUE)
Fsck's request for writing a specified block number *B* in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

4.3. Phase 1 – Check Blocks and Sizes

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format. All errors in this phase except **INCORRECT BLOCK COUNT** are fatal if the file system is being preen'ed,

CG *C*: BAD MAGIC NUMBER The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed.

UNKNOWN FILE TYPE I=*I* (CLEAR) The mode word of the inode *I* indicates that the inode is not a special block inode, special character inode, socket inode, regular inode, symbolic link, or directory inode.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO ignore this error condition.

LINK COUNT TABLE OVERFLOW (CONTINUE)
An internal table for *fsck* containing allocated inodes with a link count of zero has no more room. Recompile *fsck* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO terminate the program.

B* BAD I=*I
Inode *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the **EXCESSIVE BAD BLKS** error condition in Phase 1 if inode *I* has too many block numbers outside the file system range. This error condition will always invoke the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE BAD BLKS I=*I* (CONTINUE)
There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the

file system associated with inode *I*.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

B **DUP I=***I*

Inode *I* contains block number *B* which is already claimed by another inode. This error condition may invoke the **EXCESSIVE DUP BLKS** error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE DUP BLKS I=*I* **(CONTINUE)**

There is more than a tolerable number (usually 10) of blocks claimed by other inodes.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

DUP TABLE OVERFLOW (CONTINUE)

An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

NO terminate the program.

PARTIALLY ALLOCATED INODE I=*I* **(CLEAR)**

Inode *I* is neither allocated nor unallocated.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

INCORRECT BLOCK COUNT I=*I* **(***X* **should be** *Y***) (CORRECT)**

The block count for inode *I* is *X* blocks, but should be *Y* blocks. When preen'ing the count is corrected.

Possible responses to the CORRECT prompt are:

YES replace the block count of inode *I* with *Y*.

NO ignore this error condition.

4.4. Phase 1B: Rescan for More Dups

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This section lists the error condition when the duplicate block is found.

B **DUP I=***I*
Inode *I* contains block number *B* that is already claimed by another inode. This error condition will always invoke the **BAD/DUP** error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1.

4.5. Phase 2 – Check Pathnames

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes. All errors in this phase are fatal if the file system is being preen'ed.

ROOT INODE UNALLOCATED. TERMINATING.
The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate.

NAME TOO LONG *F*
An excessively long path name has been found. This is usually indicative of loops in the file system name space. This can occur if the super user has made circular links to directories. The offending links must be removed (by a guru).

ROOT INODE NOT DIRECTORY (FIX)
The root inode (usually inode number 2) is not directory inode type.

Possible responses to the FIX prompt are:

YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.

NO terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to the CONTINUE prompt are:

YES ignore the **DUPS/BAD** error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in a large number of other error conditions.

NO terminate the program.

I OUT OF RANGE I=*I* **NAME=***F* **(REMOVE)**
A directory entry *F* has an inode number *I* which is greater than the end of the inode list.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=*I* **OWNER=***O* **MODE=***M* **SIZE=***S* **MTIME=***T* **DIR=***F* **(REMOVE)**
A directory entry *F* has a directory inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* FILE=*F* (REMOVE)
A directory entry *F* has an inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, directory inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* FILE=*F* (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

ZERO LENGTH DIRECTORY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
A directory entry *F* has a size *S* that is zero. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed; this will always invoke the BAD/DUP error condition in Phase 4.

NO ignore this error condition.

DIRECTORY TOO SHORT I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *F* has been found whose size *S* is less than the minimum size directory. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the FIX prompt are:

YES increase the size of the directory to the minimum directory size.

NO ignore this directory.

DIRECTORY CORRUPTED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (SALVAGE)
A directory with an inconsistent internal state has been found.

Possible responses to the FIX prompt are:

YES throw away all entries up to the next 512-byte boundary. This rather drastic action can throw away up to 42 entries, and should be taken only after other recovery efforts have failed.

NO Skip up to the next 512-byte boundary and resume reading, but do not modify the directory.

BAD INODE NUMBER FOR '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose inode number for '.' does does not equal *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '.' to be equal to *I*.

NO leave the inode number for '.' unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose first entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '.' with inode number equal to *I*.

NO leave the directory unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, FIRST ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose first entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '.'
A directory *I* has been found whose first entry is not '.'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '.' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '.'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '.'.

NO leave the directory unchanged.

BAD INODE NUMBER FOR '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose inode number for '..' does does not equal the parent of *I*.

Possible responses to the FIX prompt are:

YES change the inode number for '..' to be equal to the parent of *I*.

NO leave the inode number for '..' unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose second entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '..' with inode number equal to the parent of *I*.

NO leave the directory unchanged.

**MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, SECOND ENTRY IN DIRECTORY CONTAINS *F***
A directory *I* has been found whose second entry is *F*. *Fsck* cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

**MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '..'**
A directory *I* has been found whose second entry is not '..'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '..' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '..'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '..'.

NO leave the directory unchanged.

4.6. Phase 3 – Check Connectivity

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
The directory inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing, the directory is reconnected if its size is non-zero, otherwise it is cleared.

Possible responses to the RECONNECT prompt are:

YES reconnect directory inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode *I* to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

NO ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck*(8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck*(8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

DIR I=*I1* CONNECTED. PARENT WAS I=*I2*
This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory.

4.7. Phase 4 – Check Reference Counts

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, symbolic links, or special files, unreferenced files, symbolic links, and directories, bad and duplicate blocks in files, symbolic links, and directories, and incorrect total free-inode counts. All errors in this phase are correctable if the file system is being preen'ed except running out of space in the lost+found directory.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing the file is cleared if either its size or its link count is zero, otherwise it is reconnected.

Possible responses to the RECONNECT prompt are:

YES reconnect inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found.*

NO ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

(CLEAR)
The inode mentioned in the immediately previous error condition can not be reconnected. This cannot occur if the file system is being preen'ed, since lack of space to reconnect files is a fatal error.

Possible responses to the CLEAR prompt are:

YES de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.

NO ignore this error condition.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found.* This will always invoke the CLEAR error condition in Phase 4. Check access modes of *lost+found.* This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found.* This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found.* This error is fatal if the file system is being preen'ed.

LINK COUNT FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for inode *I* which is a file, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode *I* with *Y*.

NO ignore this error condition.

LINK COUNT DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for inode *I* which is a directory, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of directory inode *I* with *Y*.

NO ignore this error condition.

LINK COUNT *F* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for *F* inode *I* is *X* but should be *Y*. The name *F*, owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of inode *I* with *Y*.

NO ignore this error condition.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Inode *I* which is a file, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing, this is a file that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Inode *I* which is a directory, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing, this is a directory that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

BAD/DUP FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

BAD/DUP DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode

I. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX)
The actual count of the free inodes does not match the count in the super-block of the file system. When preen'ing, the count is fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.8. Phase 5 - Check Cyl groups

This phase concerns itself with the free-block maps. This section lists error conditions resulting from allocated blocks in the free-block maps, free blocks missing from free-block maps, and the total free-block count incorrect.

CG *C*: BAD MAGIC NUMBER
The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually the cylinder group is marked as needing to be reconstructed. This error is fatal if the file system is being preen'ed.

EXCESSIVE BAD BLKS IN BIT MAPS (CONTINUE)
An inode contains more than a tolerable number (usually 10) of blocks claimed by other inodes or that are out of the legal range for the file system. This error is fatal if the file system is being preen'ed.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free-block maps and continue the execution of *fsck*.

NO terminate the program.

SUMMARY INFORMATION *T* BAD
where *T* is one or more of:
(INODE FREE)
(BLOCK OFFSETS)
(FRAG SUMMARIES)
(SUPER BLOCK SUMMARIES)
The indicated summary information was found to be incorrect. This error condition will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the summary information is recomputed.

***X* BLK(S) MISSING**
X blocks unused by the file system were not found in the free-block maps. This error condition will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the block maps are rebuilt.

BAD CYLINDER GROUPS (SALVAGE)
Phase 5 has found bad blocks in the free-block maps, duplicate blocks in the free-block maps, or blocks missing from the file system. When preen'ing, the cylinder groups are reconstructed.

Possible responses to the SALVAGE prompt are:

YES replace the actual free-block maps with a new free-block maps.

NO ignore this error condition.

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)
The actual count of free blocks does not match the count in the super-block of the file system. When preen'ing, the counts are fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.9. Phase 6 - Salvage Cylinder Groups

This phase concerns itself with the free-block maps reconstruction. No error messages are produced.

4.10. Cleanup

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

***V* files, *W* used, *X* free (*Y* frags, *Z* blocks)**
This is an advisory message indicating that the file system checked contained *V* files using *W* fragment sized blocks leaving *X* fragment sized blocks free in the file system. The numbers in parenthesis breaks the free count down into *Y* free fragments and *Z* free full sized blocks.

******* REBOOT UNIX *******
This is an advisory message indicating that the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps. When preen'ing, *fsck* will exit with a code of 4. The auto-reboot script interprets an exit code of 4 by issuing a reboot system call.

******* FILE SYSTEM WAS MODIFIED *******
This is an advisory message indicating that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 4.2

Sendmail implements a general purpose internetwork mail routing facility under the UNIX* operating system. It is not tied to any one transport protocol - its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for *sendmail*, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The appendixes give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail – An Internetwork Mail Router.* This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

1. BASIC INSTALLATION

There are two basic steps to installing sendmail. The hard part is to build the configuration table. This is a file that sendmail reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of sendmail assuming you can use one of the existing configurations and that the standard installation parameters are

*UNIX is a trademark of Bell Laboratories.

acceptable. All pathnames and examples are given from the root of the *sendmail* subtree.

1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectory *cf* of the sendmail directory. The ones used at Berkeley are in *m4*(1) format; files with names ending ".m4" are *m4* include files, while files with names ending ".mc" are the master files. Files with names ending ".cf" are the *m4* processed versions of the corresponding ".mc" file.

Two off the shelf configuration files are supplied to handle the basic cases: *cf/arpaproto.cf* for Arpanet (TCP) sites and *cf/uucpproto.cf* for UUCP sites. These are *not* in *m4* format. The file you need should be copied to a file with the same name as your system, e.g.,

```
cp uucpproto.cf ucsfcgl.cf
```

This file is now ready for installation as */usr/lib/sendmail.cf*.

1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.2BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should already have created your configuration file and left it in the file "cf/*system*.cf" where *system* is the name of your system (i.e., what is returned by *hostname*(1)). If you do not have *hostname* you can use the declaration "HOST=*system*" on the *make*(1) command line. You should also examine the file *md/config.m4* and change the *m4* macros there to reflect any libraries and compilation flags you may need.

The basic installation procedure is to type:

```
make
make install
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second *make* command must be executed as the superuser (root).

1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

1.3.1. lib/libsys.a

The library in lib/libsys.a contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the new 4.2BSD directory code and do not have the compatibility routines installed in your system library, you should execute the commands:

```
cd lib
make ndir
```

This will compile and install the 4.2 compatibility routines in the library. You should then type:

```
cd lib      # if required
make
```

This will recompile and fill the library.

1.3.2. /usr/lib/sendmail

The binary for sendmail is located in /usr/lib. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:

```
cd src
rm -f *.o
make
cp sendmail /usr/lib
```

1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail*:

```
rm -f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

1.3.5. /usr/lib/sendmail.cf

The configuration file must be installed in /usr/lib. This is described above.

1.3.6. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run setuid, when *mqueue* should be owned by the sendmail owner and mode 755.

1.3.7. /usr/lib/aliases*

The system aliases are held in three files. The file "/usr/lib/aliases" is the master copy. A sample is given in "lib/aliases" which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm*(3) routines. These are stored in "/usr/lib/aliases.dir" and "/usr/lib/aliases.pag." These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

1.3.8. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/sendmail.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail -bz
```

1.3.9. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to "/etc/rc" (or "/etc/rc.local" as appropriate) in the area where it is starting up the daemons:

```
if [ -f /usr/lib/sendmail ]; then
        (cd /usr/spool/mqueue; rm -f [lnx]f*)
        /usr/lib/sendmail -bd -q30m &
        echo -n ' sendmail' >/dev/console
fi
```

The "cd" and "rm" commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: "-bd" causes it to listen on the SMTP port, and "-q30m" causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the **-bd** flag.

1.3.10. /usr/lib/sendmail.hf

This is the help file used by the SMTP **HELP** command. It should be copied from "lib/sendmail.hf":

```
cp lib/sendmail.hf /usr/lib
```

1.3.11. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file "/usr/lib/sendmail.st":

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program "aux/mailstats."

1.3.12. /etc/syslog

You may want to run the *syslog* program (to collect log information about sendmail). This program normally resides in */etc/syslog*, with support files */etc/syslog.conf* and */etc/syslog.pid*. The program is located in the *aux* subdirectory of the *sendmail* distribution. The file */etc/syslog.conf* describes the file(s) that sendmail will log in. For a complete description of syslog, see the manual page for *syslog*(8) (located in *sendmail/doc* on the distribution).

1.3.13. /usr/ucb/newaliases

If *sendmail* is invoked as "newaliases," it will simulate the **–bi** flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

1.3.14. /usr/ucb/mailq

If *sendmail* is invoked as "mailq," it will simulate the **–bp** flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

2. NORMAL OPERATIONS

2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the **–bz** flag:

/usr/lib/sendmail -bz

This creates the file */usr/lib/sendmail.fc* ("frozen configuration"). This file is an image of *sendmail*'s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf* *sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.

The frozen configuration file will be ignored if a **–C** flag is specified or if sendmail detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

2.2. The System Log

The system log is supported by the *syslog*(8) program.

2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

2.2.2. Levels

If you have *syslog*(8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.3.

2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

2.3.1. Printing the queue

The contents of the queue can be printed using the *mailq* command (or by specifying the **–bp** flag to sendmail):

mailq

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

2.3.2. Format of queue files

All queue files have the form *xfAA99999* where *AA99999* is the *id* for this file and the *x* is a type. The types are:

d The data file. The message body (excluding the header) is kept in this file.

l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous **lf** file can cause a job to apparently disappear (it will not even time out!).

n This file is created when an id is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.

q The queue control file. This file contains the information necessary to process the job.

t A temporary file. These are an image of the **qf** file when it is being rebuilt. It should be renamed to a **qf** file very quickly.

x A transcript file, existing during the life of a session showing everything that happens during that session.

The **qf** file is structured as a series of lines each beginning with a code letter. The lines are as follows:

D The name of the data file. There may only be one of these lines.

H A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.

R A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.

S The sender address. There may only be one of these lines.

T The job creation time. This is used to compute when to time out the job.

P The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority increases as the message sits in the queue. The initial priority depends on the message class and the size of the message.

M A message. This line is printed by the *mailq* command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to "mckusick@calder" and "wnj":

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
        id A13557; 23-Oct-82 15:49:32-PDT (Sat)
Hphone: (415) 548-3211
HTo: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

2.3.3. Forcing the queue

Sendmail should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it ignores the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

```
cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue
```

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

```
/usr/lib/sendmail -oQ/usr/spool/omqueue -q
```

The **–oQ** flag specifies an alternate queue directory and the **–q** flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the **–v** flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

```
rmdir /usr/spool/omqueue
```

2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

name: name1, name2, ...

Only local names may be aliased; e.g.,

eric@mit-xx: eric@berkeley

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign ("#") are comments.

The second form is processed by the *dbm*(3) library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

newaliases

This is equivalent to giving *sendmail* the **–bi** flag:

/usr/lib/sendmail -bi

If the "D" option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under which it will do this are:

(1) The DBM version of the database is mode 666. -or-

(2) *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before sendmail will access the database, it checks to insure that this entry exists[1]. It will wait up to five minutes for this entry to appear, at which point it will force a rebuild itself[2].

---

[1]The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail*.

[2]Note: the "D" option must be specified in the configuration file for this operation to occur.

2.4.3. List owners

If an error occurs on sending to a certain address, say "*x*", *sendmail* will look for an alias of the form "owner-*x*" to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintanence of the list itself; in this case the list maintainer would be the owner of the list. For example:

```
unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,
        sam@matisse
owner-unix-wizards: eric@ucbarpa
```

would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.

2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

```
mckusick@ernie
kirk@calder
```

then any mail arriving for "mckusick" will be redirected to the specified accounts.

2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete. if the mailer has the **l** flag (local delivery) set in the mailer descriptor.

2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Appendix A. Some important arguments are described here.

3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the **–q** flag. If you run in mode **f** or **a** this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in **q** mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the **–bd** flag. The **–bd** flag and the **–q** flag may be combined in one call:

```
/usr/lib/sendmail -bd -q30m
```

3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the **–q** flag (with no value). It is entertaining to use the **–v** flag (verbose) when this is done to watch what happens:

```
/usr/lib/sendmail -q -v
```

3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more information. The convention is that levels greater than nine are "absurd," i.e., they print out so much information that you wouldn't normally want to see them except for debugging that particular piece of code. Debug flags are set using the **–d** option; the syntax is:

```
debug-flag:   –d debug-list
debug-list:   debug-option [ , debug-option ]
debug-option: debug-range [ . debug-level ]
debug-range:  integer | integer - integer
debug-level:  integer
```

where spaces are for reading ease only. For example,

```
-d12        Set flag 12 to level 1
-d12.3      Set flag 12 to level 3
-d3-17      Set flags 3 through 17 to level 1
-d3-17.4    Set flags 3 through 17 to level 4
```

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the **–C** flag; for example,

```
/usr/lib/sendmail -Ctest.cf
```

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the **–C** flag has no value it defaults to *sendmail.cf* in the current directory.

3.6. Changing the Values of Options

Options can be overridden using the **–o** flag. For example,

/usr/lib/sendmail -oT2m

sets the **T** (timeout) option to two minutes for this run only.

4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line "OT3d" sets option "T" to the value "3d" (three days).

4.1. Timeouts

All time intervals are set using a scaled syntax. For example, "10m" represents ten minutes, whereas "2h30m" represents two and a half hours. The full set of scales is:

s seconds
m minutes
h hours
d days
w weeks

4.1.1. Queue interval

The argument to the **–q** flag specifies how often a subdaemon will run the queue. This is typically set to between five minutes and one half hour.

4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the **r** option in the configuration file.

4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the **T** option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

/usr/lib/sendmail -oT1d -q

will run the queue and flush anything that is one day old.

4.2. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the "d" configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

i deliver interactively (synchronously)
b deliver in background (asynchronously)
q queue only (don't deliver)

There are tradeoffs. Mode "i" passes the maximum amount of information to the

sender, but is hardly ever necessary. Mode "q" puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode "b" is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

4.3. Log Level

The level of logging can be set for sendmail. The default using a standard configuration table is level 9. The levels are as follows:

0 No logging.

1 Major problems only.

2 Message collections and failed deliveries.

3 Successful deliveries.

4 Messages being defered (due to a host being down, etc.).

5 Normal message queueups.

6 Unusual but benign incidents, e.g., trying to process a locked queue file.

9 Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts.

12 Several messages that are basically only of interest when debugging.

16 Verbose information regarding the queue.

4.4. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

4.4.1. To suid or not to suid?

Sendmail can safely be made setuid to root. At the point where it is about to *exec*(2) a mailer, it checks to see if the userid is zero; if so, it resets the userid and groupid to a default (set by the **u** and **g** options). (This can be overridden by setting the **S** flag to the mailer for mailers that are trusted and must be called as root.) However, this will cause mail processing to be accounted (using *sa*(8)) to root rather than to the user sending the mail.

4.4.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the "F" option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

4.4.3. Should my alias database be writable?

At Berkeley we have the alias database (/usr/lib/aliases*) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can "steal" any other user's mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in /usr/lib). The mode on these files should match the mode on /usr/lib/aliases. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to

the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have auto-rebuild enabled (with the "D" option), then you must be careful to reconstruct the alias database each time you change the text version:

newaliases

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the "future project" list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol ('#') are comments.

5.1.1. R and S – rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and may be referenced by the mailer definitions or by other rewriting sets.

The syntax of these two commands are:

S*n*

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

R*lhs rhs comments*

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

5.1.2. D – define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

D*xval*

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence **$***x*.

5.1.3. C and F – define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

C*cword1 word2...*
F*cfile* [*format*]

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

CHmonet ucbmonet

and

CHmonet
CHucbmonet

are equivalent. The second form reads the elements of the class *c* from the named *file*; the *format* is a *scanf*(3) pattern that should produce a single string.

5.1.4. M – define mailer

Programs and interfaces to mailers are defined in this line. The format is:

M*name*, {*field*=*value*}*

where *name* is the name of the mailer (used internally only) and the "field=name" pairs define attributes of the mailer. Fields are:

Path	The pathname of the mailer
Flags	Special flags for this mailer
Sender	A rewriting set for sender addresses
Recipient	A rewriting set for recipient addresses
Argv	An argument vector to pass to this mailer
Eol	The end-of-line string for this mailer
Maxsize	The maximum message length to this mailer

Only the first character of the field name is checked.

5.1.5. H – define header

The format of the header lines that sendmail inserts into the message are defined by the **H** line. The syntax of this line is:

H[**?***mflags***?**]*hname*: *htemplate*

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

5.1.6. O – set option

There are a number of "random" options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

O*ovalue*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t", "T", "f", or "F"; the default is TRUE), or a time interval.

5.1.7. T – define trusted users

Trusted users are those users who are permitted to override the sender address using the **–f** flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

T*user1 user2...*

There may be more than one of these lines.

5.1.8. P – precedence definitions

Values for the "Precedence:" field may be defined using the **P** control line. The syntax of this field is:

P*name*=*num*

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

Pfirst-class=0
Pspecial-delivery=100
Pjunk=-100

5.2. The Semantics

This section describes the semantics of the configuration file.

5.2.1. Special macros, conditionals

Macros are interpolated using the construct **$***x*, where *x* is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

The following macros *must* be defined to transmit information into *sendmail:*

e	The SMTP entry message
j	The "official" domain name for this site
l	The format of the UNIX from line
n	The name of the daemon (for error messages)
o	The set of "operators" in addresses
q	default format of sender address

The **$e** macro is printed out when SMTP starts up. The first word must be the **$j** macro. The **$j** macro should be in RFC821 format. The **$l** and **$n** macros can be considered constants except under terribly unusual circumstances. The **$o** macro

consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "r" were in the **$o** macro, then the input "address" would be scanned as three tokens: "add," "r," and "ess." Finally, the **$q** macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g  $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An acceptable alternative for the **$q** macro is "$?x$x $.<$g>". These correspond to the following two formats:

```
eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>
```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

a	The origination date in Arpanet format
b	The current date in Arpanet format
c	The hop count
d	The date in UNIX (ctime) format
f	The sender (from) address
g	The sender address relative to the recipient
h	The recipient host
i	The queue id
p	Sendmail's pid
r	Protocol used
s	Sender's host name
t	A numeric representation of the current time
u	The recipient user
v	The version number of sendmail
w	The hostname of this site
x	The full name of the sender
y	The id of the sender's tty
z	The home directory of the recipient

There are three types of dates that can be used. The **$a** and **$b** macros are in Arpanet format; **$a** is the time as extracted from the "Date:" line of the message (if there was one), and **$b** is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, **$a** is set to the current time also. The **$d** macro is equivalent to the **$a** macro in UNIX (ctime) format.

The **$f** macro is the id of the sender as originally determined; when mailing to a specific host the **$g** macro is set to the address of the sender *relative to the recipient.* For example, if I send to "bollard@matisse" from the machine "ucbarpa" the **$f** macro will be "eric" and the **$g** macro will be "eric@ucbarpa."

The **$x** macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail.* The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the **$h**, **$u**, and **$z** macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the **$@** and **$:** part of the rewriting rules, respectively.

The **$p** and **$t** macros are used to create unique strings (e.g., for the "Message-Id:" field). The **$i** macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The **$y** macro is set to the id of the terminal of the sender (if known); some systems like to put this in the Unix "From" line. The **$v** macro is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The **$w** macro is set to the name of this host if it can be determined. The **$c** field is set to the "hop count," i.e., the number of times this message has been processed. This can be determined by the **–h** flag on the command line or by counting the timestamps in the message.

The **$r** and **$s** fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

Conditionals can be specified using the syntax:

$?x text1 $| text2 $.

This interpolates *text1* if the macro **$x** is set, and *text2* otherwise. The "else" (**$|**) clause may be omitted.

5.2.2. Special classes

The class **$=w** is set to be the set of all names this host is known by. This can be used to delete local hostnames.

5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasymbols are:

$*	Match zero or more tokens
$+	Match one or more tokens
$–	Match exactly one token
$=*x*	Match any token in class *x*
$~*x*	Match any token not in class *x*

If any of these match, they are assigned to the symbol **$***n* for replacement on the right hand side, where *n* is the index in the LHS. For example, if the LHS:

$-:$+

is applied to the input:

UCBARPA:eric

the rule will match, and the values passed to the RHS will be:

$1	UCBARPA
$2	eric

5.2.4. The right hand side

When the right hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they are begin with a dollar sign. Metasymbols are:

$*n*	Substitute indefinite token *n* from LHS
$>*n*	"Call" ruleset *n*
$#*mailer*	Resolve to *mailer*
$@*host*	Specify *host*
$:*user*	Specify *user*

The **$***n* syntax substitutes the corresponding value from a **$+**, **$–**, **$***, **$=**, or **$~** match on the LHS. It may be used anywhere.

The **$>***n* syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset *n*. The final value of ruleset *n* then becomes the substitution for this rule.

The **$#** syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

$#***mailer***$@*host***$:***user*

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceeded by a **$@** or a **$:** to control evaluation. A **$@** prefix causes the ruleset to return with the remainder of the RHS as the value. A **$:** prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The **$@** and **$:** prefixes may preceed a **$>** spec; for example:

```
R$+     $:$>7$1
```

matches anything, passes that to ruleset seven, and continues; the **$:** is necessary to avoid an infinite loop.

5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

local-part@host-domain-spec

If no "@" sign is specified, then the host-domain-spec *may* be appended from the sender address (if the **C** flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by sendmail before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Appendix C. Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.

5.2.7. The "error" mailer

The mailer with the special name "error" can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

$#error$:Host unknown in this domain

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail*. This involves setting the options, defining a few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient

addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization. Finally, ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

5.3.2.1. Large site, many hosts – minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

For example, a typical site on our local ether network is "monet." Monet has a list of known ethernet hosts; if it receives mail for any of them, it can do direct delivery. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and reject the message, or may be able to do delivery. However, it is important to note that when a new ethernet host is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated as convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, our TCP connection is currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a TCP host is transfered via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to the Arpanet. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send TCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as a TCP host it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, as if ucbarpa thought that ucbvax had the TCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4*(1) or some similar tool) as any

logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

5.3.2.2. Small site – complete information

A small site (two or three hosts) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the the configuration remains relatively static, the update problem will probably not be too great.

5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

5.3.3. Relevant issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included on the *sendmail* tape as *doc/rfc819.lpr* and *doc/rfc822.lpr*.

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

< > ( ) " \

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host is "a.cc.berkeley.arpa"; reading from right to left, "arpa" is a top level domain (related to, but not limited to, the physical Arpanet), "berkeley" is both an Arpanet host and a logical domain which is actually interpreted by a host called ucbvax (which is actually just the "major" host for this domain), "cc" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center; this particular host happens to be connected via berknet, but other hosts might be connected via one of two ethernets or some other network.

Beware when reading RFC819 that there are a number of errors in it.

5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best

deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

5.3.5. Testing the rewriting rules – the –bt flag

When you build a configuration table, you can do a certain amount of testing using the "test mode" of *sendmail*. For example, you could invoke *sendmail* as:

sendmail -bt -Ctest.cf

which would read the configuration file "test.cf" and enter test mode. In this mode, you enter lines of the form:

rwset address

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of rwsets for sequential application of rules to an input; ruleset three is always applied first. For example:

1,21,4 monet:bollard

first applies ruleset three to the input "monet:bollard." Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the "-d21" flag to turn on more debugging. For example,

sendmail -bt -d21.99

turns on an incredible amount of information; a single word address is probably going to print out several pages worth of information.

5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names "local" and "prog" must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string "[IPC]" instead.

The F field defines the mailer flags. You should specify an "f" or "r" flag to pass the name of the sender as a **–f** or **–r** flag respectively. These flags are only passed if they were passed to *sendmail*, so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify "-f $g" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses

[3] *Sendmail* must be running setuid to root for this to work.

if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template containing **$u** will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run[4].

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

From: eric@ucbarpa
To: wnj@monet, mckusick

will be modified to:

From: eric@ucbarpa
To: wnj@monet, mckusick@ucbarpa

if and only if the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Appendix C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

From: eric

might be changed to be:

From: eric@ucbarpa

or

From: ucbvax!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a **$u** macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

IPC $h [*port*]

where *port* is the optional port number to connect to.

For example, the specifications:

Mlocal, P=/bin/mail, F=rlsm S=10, R=20, A=mail -d $u
Mether,P=[IPC], F=meC, S=11, R=21, A=IPC $h, M=100000

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky **-r** flag, does local

[4]The "c" configuration option must be given for this to be effective.

delivery, quotes should be stripped from addresses, and multiple users can be delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a message will be the word "mail," the word "-d," and words containing the name of the receiving user. If a **-r** flag is inserted it will be between the words "mail" and "-d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.

APPENDIX A

COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

-f *addr* — The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).

-r *addr* — An obsolete form of **-f**.

-h *cnt* — Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAXHOP (currently 30) *sendmail* throws away the message with an error.

-F*name* — Sets the full name of this user to *name*.

-n — Don't do aliasing or forwarding.

-t — Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc:" line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.

-b*x* — Set operation mode to *x*. Operation modes are:

- m Deliver mail (default)
- a Run in arpanet mode (see below)
- s Speak SMTP on input side
- d Run as a daemon
- t Run in test mode
- v Just verify addresses, don't collect or deliver
- i Initialize the alias database
- p Print the mail queue
- z Freeze the configuration file

The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.

-q*time* — Try to process the queued up mail. If the time is given, a sendmail will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.

-C*file* — Use a different configuration file.

-d*level* — Set debugging level.

-o*xvalue* — Set option *x* to the specified *value*. These options are described in Appendix B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the **-s** flag.

APPENDIX B

CONFIGURATION OPTIONS

The following options may be set using the **–o** flag on the command line or the **O** line in the configuration file:

A*file* — Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.

a — If set, wait for an "@:@" entry to exist in the alias database before starting up. If it does not appear in five minutes, rebuild the database.

c — If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.

d*x* — Deliver in mode *x*. Legal modes are:

- i Deliver interactively (synchronously)
- b Deliver in background (asynchronously)
- q Just queue the message (deliver during queue run)

D — If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **–bi**.

e*x* — Dispose of errors using mode *x*. The values for *x* are:

- p Print error messages (default)
- q No messages, just give exit status
- m Mail back errors
- w Write back errors (mail if user not logged in)
- e Mail back errors and give zero exit stat always

F*n* — The temporary file mode, in octal. 644 and 600 are good choices.

f — Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.

g*n* — Set the default group id for mailers to run in to *n*.

H*file* — Specify the help file for SMTP.

i — Ignore dots in incoming messages.

L*n* — Set the default log level to *n*.

M*xvalue* — Set the macro *x* to *value*. This is intended only for use from the command line.

m — Send to me too, even if I am in an alias expansion.

o — Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.

Q*dir*	Use the named *dir* as the queue directory.
r*time*	Timeout reads after *time* interval.
S*file*	Log statistics in the named *file*.
s	Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control the the client under any circumstances.
T*time*	Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender.
t*S,D*	Set the local timezone name to *S* for standard time and *D* for daylight time; this is only used under version six.
u*n*	Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user.
v	Run in verbose mode.

APPENDIX C

MAILER FLAGS

The following flags may be set in the mailer description.

f The mailer wants a **–f** *from* flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).

r Same as **f**, but sends a **–r** flag.

S Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).

n Do not insert a UNIX-style "From" line on the front of the message.

l This mailer is local (i.e., final delivery will be performed).

s Strip quote characters off of the address before calling the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a **$u** macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.

F This mailer wants a "From:" header line.

D This mailer wants a "Date:" header line.

M This mailer wants a "Message-Id:" header line.

x This mailer wants a "Full-Name:" header line.

P This mailer wants a "Return-Path:" line.

u Upper case should be preserved in user names for this mailer.

h Upper case should be preserved in host names for this mailer.

A This is an Arpanet-compatible mailer, and all appropriate modes should be set.

U This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.

e This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.

X This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.

L Limit the line lengths as specified in RFC821.

P Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.

I This mailer will be speaking SMTP to another *sendmail* - as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign ("@") after being rewritten by ruleset three will have the "@domain" clause from the sender tacked on. This allows mail with headers of the form:

From: usera@hosta
To: userb@hostb, userc

to be rewritten as:

From: usera@hosta
To: userb@hostb, userc@hosta

automatically.

APPENDIX D

OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

md/config.m4 These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.

src/conf.h Configuration parameters that may be tweaked by the installer are included in conf.h.

src/conf.c Some special routines and a few variables may be defined in conf.c. For the most part these are selected from the settings in conf.h.

Parameters in md/config.m4

The following compilation flags may be defined in the *m4CONFIG* macro in *md/config.m4* to define the environment in which you are operating.

V6 If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.

VMUNIX If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork*(2) system call, special types defined in <sys/types.h> (e.g, u_char), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include if you are running a 4.1BSD system.

Parameters in src/conf.h

Parameters and compilation options are defined in conf.h. Most of these need not normally be tweaked; common parameters are all in sendmail.cf. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

MAXLINE [256] The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.

MAXNAME [128] The maximum length of any name, such as a host or a user name.

MAXFIELD [2500] The maximum total length of any header field, including continuation lines.

MAXPV [40] The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.

MAXHOP [30] When a message has been processed more than this number of times, sendmail rejects the message on the assumption that there has been an aliasing loop. This can be determined from the **–h** flag or by counting the number of trace fields (i.e, "Received:" lines) in the message header.

MAXATOM [100] The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

MAXMAILERS [25]
The maximum number of mailers that may be defined in the configuration file.

MAXRWSETS [30]
The maximum number of rewriting sets that may be defined.

MAXPRIORITIES [25]
The maximum number of values for the "Precedence:" field that may be defined (using the **P** line in sendmail.cf).

MAXTRUST [30] The maximum number of trusted users that may be defined (using the **T** line in sendmail.cf).

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

DBM If set, the "DBM" package in UNIX is used (see DBM(3X) in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

DEBUG If set, debugging information is compiled in. To actually get the debugging output, the **–d** flag must be used.

LOG If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

QUEUE This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

SMTP If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

DAEMON If set, code to run a daemon is compiled in. This code is for 4.2BSD if the NVMUNIX flag is specified; otherwise, 4.1a BSD code is used. Beware however that there are bugs in the 4.1a code that make it impossible for **sendmail** to work correctly under heavy load.

UGLYUUCP If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

NOTUNIX If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

Configuration in src/conf.c

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below), The flags are:

H_ACHECK Normally when the check is made to see if a header line is compatible with a mailer, *sendmail* will not delete an existing line. If this flag is set, *sendmail* will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in sendmail.cf, the header line is *always* deleted.

H_EOH If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.

H_FORCE Add this header entry even if one existed in the message before. If a header entry does not have this bit set, *sendmail* will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.

H_TRACE If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.

H_RCPT If set, this field contains recipient addresses. This is used by the **-t** flag to determine who to send to when it is collecting recipients from the message.

H_FROM This flag indicates that this field specifies a sender. The order of these fields in the *HdrInfo* table specifies *sendmail's* preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```
struct hdrinfo          HdrInfo[] =
{
        /* originator fields, most to least significant */
    "resent-sender",    H_FROM,
    "resent-from",      H_FROM,
    "sender",           H_FROM,
    "from",             H_FROM,
    "full-name",        H_ACHECK,
        /* destination fields */
    "to",               H_RCPT,
    "resent-to",        H_RCPT,
    "cc",               H_RCPT,
        /* message identification and control */
    "message",          H_EOH,
    "text",             H_EOH,
        /* trace fields */
    "received",         H_TRACEH_FORCE,

    NULL,               0,
};
```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro **$x** and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char Arpa_Info[] =      "050"; /* arbitrary info */
char Arpa_TSyserr[] =   "455"; /* some (transient) system error */
char Arpa_PSyserr[] =   "554"; /* some (transient) system error */
char Arpa_Usrerr[] =    "554"; /* some (fatal) user error */
```

The class *Arpa_Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa_TSyserr* and *Arpa_PSyserr* is printed by the *syserr* routine. TSyserr is printed out for transient errors, whereas PSyserr is printed for permanent errors; the distinction is made based on the value of *errno*. Finally, *Arpa_Usrerr* is the result of a user error and is generated by the *usrerr* routine; these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
    register ADDRESS *to;
{
    if (MsgSize > 50000 && to->q_mailer != LocalMailer)
    {
        usrerr("Message too large for non-local delivery");
        NoReturn = TRUE;
        return (FALSE);
    }
    return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to supress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

APPENDIX E

SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

/usr/lib/sendmail
: The binary of *sendmail.*

/usr/bin/newaliases
: A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **–bi** flag.

/usr/bin/mailq
: Prints a listing of the mail queue. This program is equivalent to using the **–bp** flag to *sendmail.*

/usr/lib/sendmail.cf
: The configuration file, in textual form.

/usr/lib/sendmail.fc
: The configuration file represented as a memory image.

/usr/lib/sendmail.hf
: The SMTP help file.

/usr/lib/sendmail.st
: A statistics file; need not be present.

/usr/lib/aliases
: The textual version of the alias file.

/usr/lib/aliases.{pag,dir}
: The alias file in *dbm*(3) format.

/etc/syslog
: The program to do logging.

/etc/syslog.conf
: The configuration file for syslog.

/etc/syslog.pid
: Contains the process id of the currently running syslog.

/usr/spool/mqueue
: The directory in which the mail queue and temporary files reside.

/usr/spool/mqueue/qf*
: Control (queue) files for messages.

/usr/spool/mqueue/df*
: Data files.

/usr/spool/mqueue/lf*
: Lock files

/usr/spool/mqueue/tf*
: Temporary versions of the qf files, used during queue file rebuild.

/usr/spool/mqueue/nf*
: A file used when creating a unique id.

/usr/spool/mqueue/xf*
: A transcript of the current session.

PART 3: COMMUNICATIONS

The three articles in this part cover a range of communications topics, from general background information to detailed descriptions of program structures and protocols. They describe the interprocess communication software (this can be either interactive or batch) and *sendmail*, an internetwork mail server.

Ftp, *telnet*, and the *r*-command set are three other networking software utilities available on the ULTRIX–32m system but not mentioned in these articles. See the end of this introduction for a brief description of each.

Interprocess Communication

The first two articles describe the socket software, a set of system calls (new with the 4.2BSD distribution) used for interprocess communication. The communicating processes can be running on the same computer or on separate computers linked by the DARPA standard communication protocols.

Interprocess communication requires each process to set up one of three types of socket:

Stream socket	Communication is bidirectional, reliable, sequenced, and unduplicated.
Datagram socket	Communication is bidirectional but not promised to be reliable, sequenced, or unduplicated.
Raw socket	Communication is possible through access to underlying protocols.

"A 4.2BSD Interprocess Communication Primer" gives the format for each socket-related call and explains how to coordinate the calls to establish a connection and send and receive messages:

- *Create* a socket
- *Bind* a name to a socket
- *Connect* — initiate a connection
- *Listen* for a connect request
- *Accept* a connect request
- *Write* a message
- *Read* a message
- *Send* a message
- *Receive* a message
- *Sendto* — send a datagram message
- *Recvfrom* — receive a datagram message
- *Close* a connection

- *Shutdown* a connection
- *Select* — multiplex the transfer of messages

These commands are listed individually in the *ULTRIX–32m Programmer's Manual*. The article also tells how to use: a library of routines that manipulate addresses, server and client calls, and connectionless servers. And information on a variety of advanced topics is available for sophisticated users.

The second article, "4.2BSD Networking Implementation Notes," describes the internal structure of the interprocess communication software. This information should be useful to engineers who are developing new communication protocols and network utilities. The article explains:

- Support for multiple protocol families and addressing styles
- Structures for internal address representation
- Memory management for network functions
- Internal layering
- Protocol interfaces
- Gateways
- Routing tables
- Use of raw sockets for direct access to low level protocols
- Buffering issues
- Handling out-of-band data
- Use of trailer protocols

You can also find a description of the user interface to the interprocess communication software in Section 2.3 of the "4.2BSD System Manual" (in Volume II of this set). Prefer the more recent "4.2BSD Interprocess Communication Primer" when you find discrepancies.

Sendmail

The article by Allman, "Sendmail — An Internetwork Mail Router," offers good background information for people who install and maintain the *sendmail* utility. For actual instructions on installation, see the "Sendmail Installation and Operation Guide" in Part 2 of this volume.

Sendmail acts like a post office, enabling different networking systems to route mail between them. For example, people using the ARPANET and others using the ETHERNET can send mail to each other, and *sendmail* will cooperate with the network software at each end to make sure that the messages get through. The *sendmail* functions are transparent to people sending the messages; each sender or receiver needs to deal only with the interface to the local network used on his or her computer system.

A reading of this article is prerequisite to an understanding of the "Sendmail Installation and Operation Guide."

Standard Networking Utilities

Three other networking systems are available on ULTRIX–32m:

ftp	File transfer program (a user interface to the ARPANET)
telnet	Remote login protocol
r-commands	New networking software layered on sockets

You can find information on these utilities in the *ULTRIX–32m Programmer's Manual. Ftp* and *telnet* are utilities that prompt you for commands when you run them. Users must give appropriate passwords when accessing information on remote systems. The command descriptions listed under *ftp* and *telnet* are comprehensive.

The *r*-commands, like the interprocess communication commands, are listed individually in the *ULTRIX–32m Programmer's Manual*, because you must call each one from the shell:

rcmd	Connect, then execute a command
rcp	Remote file copy
rdump	File system dump across a network
rexec	Remote execute
rlogin	Remote login
rmt	Remote magnetic tape dump
rrestore	Restore a system from a file system dump across a network
rshell	Remote shell; provide remote execution facilities
ruptime	Show how long a remote system has been up
rwho	Show who is on a remote system

However, the *r*-command software requires trust between system users, because remote access using the *r*–commands does not require use of passwords.

A 4.2BSD Interprocess Communication Primer
DRAFT of August 31, 1984

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.2BSD release of the VAX* UNIX** system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* DEC and VAX are trademarks of Digital Equipment Corporation.
** UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important parts of 4.2BSD is the interprocess communication facilities. These facilities are the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporate many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. It is hoped that the interprocess communication facilities included in 4.2BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4.2BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the ipc facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact these facilities have been tied to the UNIX file system; either through naming, or implementation. Consequently, the ipc facilities provided in 4.2BSD have been designed as a totally independent subsystem. The 4.2BSD ipc allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to included more than the simple byte stream provided by a pipe-like entity. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

The remainder of this document is organized in four sections. Section 2 introduces the new system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the ipc facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.2BSD ipc supports two separate communication domains: the UNIX domain, and the Internet domain is used by processes which communicate using the the DARPA standard communication protocols. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the possible error conditions which are possible when operating in the Internet domain.

2.1. Socket types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes*.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

Two potential socket types which have interesting properties are the *sequenced packet* socket and the *reliably delivered message* socket. A sequenced packet socket is identical to a stream socket with the exception that record boundaries are preserved. This interface is very similar to that provided by the Xerox NS Sequenced Packet protocol. The reliably delivered message socket has similar properties to a datagram socket, but with reliable delivery. While these two socket types have been loosely defined, they are currently unimplemented in 4.2BSD. As such, in this document we will concern ourselves only with the three socket types for which support exists.

* In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified. To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use a sample call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

To obtain a particular protocol one selects the protocol number, as defined within the communication domain. For the Internet domain the available protocols are defined in *<netinet/in.h>* or, better yet, one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("tcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. The *bind* call is used to assign a name to a socket:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). In the UNIX domain names are path names while in the Internet domain names contain an Internet address and port number. If one wanted to bind the name "/dev/foo" to a UNIX domain socket, the following would be used:

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

```
bind(s, "/dev/foo", sizeof ("/dev/foo") - 1);
```

(Note how the null byte in the name is not counted as part of the name.) In binding an Internet address things become more complicated. The actual call is simple,

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

2.4. Connection establishment

With a bound socket it is possible to rendezvous with an unrelated process. This operation is usually asymmetric with one process a "client" and the other a "server". The client requests services from the server by initiating a "connection" to the server's socket. The server, when willing to offer its advertised services, passively "listens" on its socket. On the client side the *connect* call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
connect(s, "server-name", sizeof ("server-name"));
```

while in the Internet domain,

```
struct sockaddr_in server;
connect(s, &server, sizeof (server));
```

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket; c.f. section 5.4. An error is returned when the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

Many errors can be returned when a connection attempt fails. The most common are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. When connecting to a host running 4.2BSD this is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH
: These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down. In these cases the system is conservative and indicates the entire network is unreachable.

For the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
fromlen = sizeof (from);
snew = accept(s, &from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be zero.

Accept normally blocks. That is, the call to accept will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or not block on the accept call there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are useable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags may be specified as a non-zero value if one or more of the following is required:

SOF_OOB	send/receive out of band data
SOF_PREVIEW	look at data without reading
SOF_DONTROUTE	send data without routing packets

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When SOF_PREVIEW is specified with a *recv* call, any data present is returned to the user, but

treated as still "unread". That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received. Applying shutdown to a socket causes any data queued to be immediately discarded.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before, and each should have a name bound to it in order that the recipient of a message may identify the sender. To send data, the *sendto* primitive is used,

```
sendto(s, buf, buflen, flags, &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the intended recipient of the message. When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. Where information is present locally to recognize a message which may never be delivered (for instance when a network is unreachable), the call will return -1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket (i.e. no multicasting). Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket where a connect request initiates establishment of an end to end connection). Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

```
select(nfds, &readfds, &writefds, &execptfds, &timeout);
```

Select takes as arguments three bit masks, one for the set of file descriptors for which the caller wishes to be able to read data on, one for those descriptors to which data is to be written, and one for which exceptional conditions are pending. Bit masks are created by or-ing bits of the form "1 << fd". That is, a descriptor *fd* is selected if a 1 is present in the *fd*'th bit of the mask. The parameter *nfds* specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) specified in a mask.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If *timeout* is set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected. If the *select* call returns due to the timeout expiring, then a value of -1 is returned along with the error number EINTR.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.2BSD networking facilities support only the DARPA standard Internet protocols, these routines have been designed with flexibility in mind. As more communication protocols become available, we hope the same user interface will be maintained in accessing network-related address data bases. The only difference should be the values returned to the user. Since these values are normally supplied the system, users should not need to be directly aware of the communication protocol and/or naming conventions in use.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. "the *login server* on host monet". This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings is likely to vary between network architectures. For instance, it is desirable for a network to not require hosts be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

3.1. Host names

A host name to address mapping is represented by the *hostent* structure:

```
struct  hostent {
        char        *h_name;        /* official name of host */
        char        **h_aliases;    /* alias list */
        int         h_addrtype;     /* host address type */
        int         h_length;       /* length of address */
        char        *h_addr;        /* address */
};
```

The official name of the host and its public aliases are returned, along with a variable length address and address type. The routine *gethostbyname*(3N) takes a host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps host addresses into a *hostent* structure. It is possible for a host to have many addresses, all having the same name. *Gethostbyname* returns the first matching entry in the data base file */etc/hosts*; if this is unsuitable, the lower level routine *gethostent*(3N) may be used. For example, to obtain a *hostent* structure for a host on a particular network the following routine might be used (for simplicity, only Internet addresses are considered):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
struct hostent *
gethostbynameandnet(name, net)
        char *name;
        int net;
{
        register struct hostent *hp;
        register char **cp;

        sethostent(0);
        while ((hp = gethostent()) != NULL) {
                if (hp->h_addrtype != AF_INET)
                        continue;
                if (strcmp(name, hp->h_name)) {
                        for (cp = hp->h_aliases; cp && *cp != NULL; cp++)
                                if (strcmp(name, *cp) == 0)
                                goto found;
                        continue;
                }
        found:
                if (in_netof(*(struct in_addr *)hp->h_addr)) == net)
                        break;
        }
        endhostent(0);
        return (hp);
}
```

(*in_netof*(3N) is a standard routine which returns the network portion of an Internet address.)

3.2. Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct  netent {
        char        *n_name;        /* official name of net */
        char        **n_aliases;    /* alias list */
        int         n_addrtype;     /* net address type */
        int         n_net;          /* network # */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above.

3.3. Protocol names

For protocols the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct  protoent {
        char        *p_name;        /* official protocol name */
        char        **p_aliases;    /* alias list */
        int         p_proto;        /* protocol # */
};
```

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports or support multiple protocols. If either of these occurs, the higher level library routines will have to be bypassed in favor of homegrown routines similar in spirit to the "gethostbynameandnet" routine described above. A service mapping is described by the *servent* structure,

```
struct  servent {
        char        *s_name;        /* official service name */
        char        **s_aliases;    /* alias list */
        int         s_port;         /* port # */
        char        *s_proto;       /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *)0);
```

returns the service specification for a telnet server using any protocol, while the call

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

3.5. Miscellaneous

With the support routines described above, an application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile. Perhaps when the system is adapted to different network architectures the utilities will be reorganized more cleanly.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On a VAX, or machine with similar architecture, this is usually reversed. Consequently, programs are sometimes required to byte swap quantities. The

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
...
main(argc, argv)
        char *argv[];
{
        struct sockaddr_in sin;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...
        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&sin, sizeof (sin));
        bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
        sin.sin_family = hp->h_addrtype;
        sin.sin_port = sp->s_port;
        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
        ...
        if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 1. C run-time routines.

library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines other than the VAX these routines are defined as null macros.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

Client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process". We will first consider the properties of server processes, then client processes.

A server process normally listens at a well know address for service requests. Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. The Xerox Courier protocol uses the latter scheme. When using Courier, a Courier client process contacts a Courier server at the remote host and identifies the service it requires. The Courier server process then creates the appropriate server process based on a data base and "splices" the client and server together, voiding its part in the transaction. This scheme is attractive in that the Courier server process may provide a single contact point for all services, as well as carrying out the initial steps in authentication. However, while this is an attractive possibility for standardizing access to services, it does introduce a certain amount of overhead due to the intermediate process involved. Implementations which provide this type of service within the system can minimize the cost of client server rendezvous. The *portal* notion described in the "4.2BSD System Manual" embodies many of the ideas found in Courier, with the rendezvous mechanism implemented internal to the system.

4.1. Servers

In 4.2BSD most servers are accessed at well known Internet addresses or UNIX domain names. When a server is started at boot time it advertises it services by listening at a well know location. For example, the remote login server's main loop is of the form shown in Figure 2.

The first step taken by the server is look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

This definition is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker. This is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal.

```
main(argc, argv)
        int argc;
        char **argv;
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...
#ifndef DEBUG
        <<disassociate server from controlling terminal>>
#endif
        ...
        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (caddr_t)&sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                perror("rlogind: accept");
                        continue;
                }
                if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

Figure 2. Remote login server.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        if (fork() == 0) {
                close(f);
                doit(g, &from);
        }
        close(g);
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established. With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queueing connection requests is closed in the child, while the socket created as a result of the accept is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

4.2. Clients

The client side of the remote login service was shown earlier in Figure 1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides:

```
bzero((char *)&sin, sizeof (sin));
bcopy(hp->h_addr, (char *)sin.sin_addr, hp->h_length);
sin.sin_family = hp->h_addrtype;
sin.sin_port = sp->s_port;
```

A socket is created, and a connection initiated.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
 ...
if (connect(s, (char *)&sin, sizeof (sin)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the "rwho" service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime*(1) program. The output generated is illustrated in Figure 3.

```
arpa      up   9:45,      5 users, load  1.15,  1.39,  1.31
cad       up   2+12:04,   8 users, load  4.67,  5.13,  4.59
calder    up   10:10,     0 users, load  0.27,  0.15,  0.14
dali      up   2+06:28,   9 users, load  1.04,  1.20,  1.65
degas     up   25+09:48,  0 users, load  1.49,  1.43,  1.41
ear       up   5+00:05,   0 users, load  1.51,  1.54,  1.56
ernie   down   0:24
esvax   down   17:04
ingres  down   0:26
kim       up   3+09:16,   8 users, load  2.03,  2.46,  3.11
matisse   up   3+06:18,   0 users, load  0.03,  0.03,  0.05
medea     up   3+09:39,   2 users, load  0.35,  0.37,  0.50
merlin  down   19+15:37
miro      up   1+07:20,   7 users, load  4.59,  3.28,  2.12
monet     up   1+00:43,   2 users, load  0.22,  0.09,  0.07
oz      down   16:09
statvax   up   2+15:57,   3 users, load  1.52,  1.81,  1.86
ucbvax    up   9:34,      2 users, load  6.08,  5.16,  3.28
```

Figure 3. ruptime output.

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update

a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        bind(s, &sin, sizeof (sin));
        ...
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0, &from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                perror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from %x\n",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, FWRONLYFCREATEFTRUNCATE, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet does,

however, indicates some problems with the current protocol.

Status information is broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status received). This, unfortunately, requires some bootstrapping information, as a server started up on a quiet network will have no known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

The second problem with the current scheme is that the rwho process services only a single local network, and this network is found by reading a file. It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.2BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information†. Unfortunately, no straightforward mechanism currently exists for finding the collection of networks to which a host is directly connected. Thus the rwho server performs a lookup in a file to find its local network. A better, though still unsatisfactory, scheme used by the routing process is to interrogate the system data structures to locate those directly connected networks. A mechanism to acquire this information from the system would be a useful addition.

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.
† An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the ipc the mechanisms already described will suffice in constructing distributed applications. However, others will find need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data along with the SIGURG signal. In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals from between client and server processes. When a signal is expected to flush any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

The stream abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data) the system extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

To send an out of band message the SOF_OOB flag is supplied to a *send* or *sendto* calls, while to receive out of band data SOF_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK ioctl is provided:

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5.

5.2. Signals and process groups

Due to the existence of the SIGURG and SIGIO signals each socket has an associated process group (just as is done for terminals). This process group is initialized to the process group of its creator, but may be redefined at a later time with the SIOCSPGRP ioctl:

```
oob()
{
        int out = 1+1;
        char waste[BUFSIZ], mark;

        signal(SIGURG, oob);
        /* flush local terminal input and output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        recv(rem, &mark, 1, SOF_OOB);
        ...
}
```

Figure 5. Flushing terminal i/o on receipt of out of band data.

```
ioctl(s, SIOCSPGRP, &pgrp);
```

A similar ioctl, SIOCGPGRP, is available for determining the current process group of a socket.

5.3. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since a socket is not a terminal, it is often necessary to have a process communicating over the network do so through a *pseudo terminal.* A pseudo terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side. The remote login server uses pseudo terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends an interrupt or quit signal to a process executing on a remote machine, the client login program traps the signal, sends an out of band message to the server process who then uses the signal number, sent as the data value in the out of band message, to perform a *killpg*(2) on the appropriate process group.

5.4. Internet address binding

Binding addresses to sockets in the Internet domain can be fairly complex. Communicating processes are bound by an *association.* An association is composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each Internet protocol. Associations are always unique. That is, there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The bind system call allows a process to specify half of an association, <local address, local port>, while the connect and accept primitives are used to complete a socket's association. Since the association is created in two steps the association uniqueness requirement

indicated above could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and addressed to any of the possible addresses assigned a host. For example, if a host is on a networks 46 and 10 and a socket is bound as above, then an accept call is performed, the process will be able to accept connection requests which arrive either from network 46 or network 10.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example:

```
sin.sin_addr.s_addr = MYADDRESS;
sin.sin_port = 0;
bind(s, (char *)&sin, sizeof (sin));
```

The system selects the port number based on two criteria. The first is that ports numbered 0 through 1023 are reserved for privileged users (i.e. the super user). The second is that the port number is not currently bound to some other socket. In order to find a free port number in the privileged range the following code is used by the remote shell server:

```
struct sockaddr_in sin;
 ...
lport = IPPORT_RESERVED - 1;
sin.sin_addr.s_addr = INADDR_ANY;
 ...
for (;;) {
        sin.sin_port = htons((u_short)lport);
        if (bind(s, (caddr_t)&sin, sizeof (sin)) >= 0)
                break;
        if (errno != EADDRINUSE && errno != EADDRNOTAVAIL) {
                perror("socket");
                break;
        }
        lport--;
        if (lport == IPPORT_RESERVED/2) {
                fprintf(stderr, "socket: All ports in use\n");
                break;
        }
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is due to associations being created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket were around. To override the default port selection algorithm then an option call must be performed prior to address binding:

```
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind(s, (char *)&sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port (if an association already exists, the error EADDRINUSE is returned).

Local address binding by the system is currently done somewhat haphazardly when a host is on multiple networks. Logically, one would expect the system to bind the local address associated with the network through which a peer was communicating. For instance, if the local host is connected to networks 46 and 10 and the foreign host is on network 32, and traffic from network 32 were arriving via network 10, the local address to be bound would be the host's address on network 10, not network 46. This unfortunately, is not always the case. For reasons too complicated to discuss here, the local address bound may be appear to be chosen at random. This property of local address binding will normally be invisible to users unless the foreign host does not understand how to reach the address selected*.

* For example, if network 46 were unknown to the host on network 32, and the local address were bound to that located on network 46, then even though a route between the two hosts existed through network 10, a connection would fail.

5.5. Broadcasting and datagram sockets

By using a datagram socket it is possible to send broadcast packets on many networks supported by the system (the network itself must support the notion of broadcasting; the system provides no broadcast simulation in software). Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to the super user.

To send a broadcast message, an Internet datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind(s, (char *)&sin, sizeof (sin));
```

Then the message should be addressed as:

```
dst.sin_family = AF_INET;
dst.sin_addr.s_addr = INADDR_ANY;
dst.sin_port = DESTPORT;
```

and, finally, a sendto call may be used:

```
sendto(s, buf, buflen, 0, &dst, sizeof (dst));
```

Received broadcast messages contain the senders address and port (datagram sockets are anchored before a message is allowed to go out).

5.6. Signals

Two new signals have been added to the system which may be used in conjunction with the interprocess communication facilities. The SIGURG signal is associated with the existence of an "urgent condition". The SIGIO signal is used with "interrupt driven i/o" (not presently implemented). SIGURG is currently supplied a process when out of band data is present at a socket. If multiple sockets have out of band data awaiting delivery, a select call may be used to determine those sockets with such data.

An old signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any children processes have changed state. Normally servers use the signal to "reap" child processes after exiting. For example, the remote login server loop shown in Figure 2 may be augmented as follows:

```
int reaper();
 ...
sigset(SIGCHLD, reaper);
listen(f, 10);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, &from, &len, 0);
        if (g < 0) {
                if (errno != EINTR)
                        perror("rlogind: accept");
                continue;
        }
        ...
}
 ...
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

PART 4: SECURITY CONSIDERATIONS

Security on the ULTRIX–32m system is flexible and reasonably comprehensive. These two articles describe a number of measures you can take to make your installation moderately secure. The first article in this part, "On the Security of UNIX," explains the major features and weaknesses of ULTRIX–32m system security. The second article, "Password Security: A Case History," tells how the password facility used on the ULTRIX–32m system was developed.

Protection Against Crashes and Unauthorized Access

Unrestricted use of disk space on the ULTRIX–32m system may cripple or stop the operating system, and Ritchie indicates in "On the Security of UNIX" that the software cannot be protected from this type of abuse. However, the ULTRIX–32m system does include the *quota* utility, which enables the system manager to control the use of resources by limiting the number of blocks and the number of files available to each user. See "Disk Quotas in a UNIX Environment" in Part 2 of this volume.

The Ritchie article explains the functions of the file protection bits, the user identification number (UID), and the user-group identification number (GID); these functions allow users to control access to their files.

In addition, Ritchie outlines the ULTRIX–32m system schemes for:

- Data encryption
- Password security
- Precautions concerning the super user account, set–UID programs, mail, and the *mount* command

Password Security Development

"Password Security: A Case History," by Morris and Thompson, outlines the objectives of the password system on the ULTRIX–32m system:

- To protect the system against invasion by unauthorized users
- To prevent logged-in users from performing unauthorized functions
- To minimize inconvenience to legitimate users

The case history describes the early, rejected password schemes and their weaknesses, and it shows how they evolved to the current scheme. You may find the article useful as well as interesting, because it offers well-researched precautions to users and tested recommendations to system managers.

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *4.2BSD System Manual* [Joy82a]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

---

†UNIX is a trademark of Bell Laboratories

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short       sa_family;      /* data format identifier */
        char        sa_data[14];    /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats*.

* Later versions of the system support variable length addresses.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*'s. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;     /* next buffer in chain */
        u_long      m_off;            /* offset of data */
        short       m_len;            /* amount of data in this mbuf */
        short       m_type;           /* mbuf type (accounting) */
        u_char      m_dat[MLEN];      /* data storage */
        struct      mbuf *m_act;      /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define   mtod(x,t)          ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);
: The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);
: The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);
: The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);
: After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is

guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define     dtom(x)     ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

6.1. Socket layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *4.2BSD System Manual* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short       so_type;            /* generic type */
        short       so_options;         /* from socket call */
        short       so_linger;          /* time to linger while closing */
        short       so_state;           /* internal state flags */
        caddr_t     so_pcb;             /* protocol control block */
        struct      protosw *so_proto;/* protocol handle */
        struct      socket *so_head;    /* back pointer to accept socket */
        struct      socket *so_q0;      /* queue of partial connections */
        short       so_q0len;           /* partials on so_q0 */
        struct      socket *so_q;       /* queue of incoming connections */
        short       so_qlen;            /* number of connections on so_q */
        short       so_qlimit;          /* max number queued connections */
        struct      sockbuf so_snd;     /* send queue */
        struct      sockbuf so_rcv;     /* receive queue */
        short       so_timeo;           /* connection timeout */
        u_short     so_error;           /* error affecting connection */
        short       so_oobmark;         /* chars to oob mark */
        short       so_pgrp;            /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the

socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

6.1.1. Socket state

A socket's state is defined from the following:

```
#define SS_NOFDREF          0x001   /* no file table ref any more */
#define SS_ISCONNECTED      0x002   /* socket connected to a peer */
#define SS_ISCONNECTING     0x004   /* in process of connecting to peer */
#define SS_ISDISCONNECTING          0x008/* in process of disconnecting */
#define SS_CANTSENDMORE     0x010   /* can't send more data to peer */
#define SS_CANTRCVMORE      0x020   /* can't receive more data from peer */
#define SS_CONNAWAITING     0x040   /* connections awaiting acceptance */
#define SS_RCVATMARK        0x080   /* at mark on input */

#define SS_PRIV             0x100   /* privileged */
#define SS_NBIO             0x200   /* non-blocking ops */
#define SS_ASYNC            0x400   /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULD-BLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        short       sb_cc;          /* actual chars in buffer */
        short       sb_hiwat;       /* max actual char count */
        short       sb_mbcnt;       /* chars of mbufs used */
        short       sb_mbmax;       /* max chars of mbufs to use */
        short       sb_lowat;       /* low water mark */
        short       sb_timeo;       /* timeout */
        struct      mbuf *sb_mb;    /* the mbuf chain */
        struct      proc *sb_sel;   /* process selecting read/write */
        short       sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define  SB_LOCK    0x01   /* lock on data queue (so_rcv only) */
#define  SB_WANT    0x02   /* someone is waiting to lock */
#define  SB_WAIT    0x04   /* someone is waiting for data/space */
#define  SB_SEL     0x08   /* buffer is selected */
#define  SB_COLL    0x10   /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket connection queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

6.2. Protocol layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short       pr_type;            /* socket type used for */
        short       pr_family;          /* protocol family */
        short       pr_protocol;        /* protocol number */
        short       pr_flags;           /* socket visible attributes */
/* protocol-protocol hooks */
        int         (*pr_input)();      /* input to protocol (from below) */
        int         (*pr_output)();     /* output to protocol (from above) */
        int         (*pr_ctlinput)();   /* control input (from below) */
        int         (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
        int         (*pr_usrreq)();     /* user request */
/* utility hooks */
        int         (*pr_init)();       /* initialization routine */
        int         (*pr_fasttimo)();   /* fast timeout (200ms) */
        int         (*pr_slowtimo)();   /* slow timeout (500ms) */
        int         (*pr_drain)();      /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define  PR_ATOMIC        0x01   /* exchange atomic messages only */
#define  PR_ADDR          0x02   /* addresses given with messages */
#define  PR_CONNREQUIRED         0x04/* connection required by protocol */
#define  PR_WANTRCVD      0x08   /* want PRU_RCVD calls */
#define  PR_RIGHTS        0x10   /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char        *if_name;          /* name, e.g. "en" or "lo" */
        short       if_unit;           /* sub-unit for lower level driver */
        short       if_mtu;            /* maximum transmission unit */
        int         if_net;            /* network number of interface */
        short       if_flags;          /* up/down, broadcast, etc. */
        short       if_timer;          /* time 'til if_watchdog called */
        int         if_host[2];        /* local net host number */
        struct      sockaddr if_addr;  /* address of interface */
        union {
                    struct             sockaddr ifu_broadaddr;
                    struct             sockaddr ifu_dstaddr;
        } if_ifu;
        struct      ifqueue if_snd;    /* output queue */
        int         (*if_init)();      /* init routine */
        int         (*if_output)();    /* output routine */
        int         (*if_ioctl)();     /* ioctl routine */
        int         (*if_reset)();     /* bus reset routine */
        int         (*if_watchdog)();  /* timer routine */
        int         if_ipackets;       /* packets received on interface */
        int         if_ierrors;        /* input errors on interface */
        int         if_opackets;       /* packets sent on interface */
        int         if_oerrors;        /* output errors on interface */
        int         if_collisions;     /* collisions on csma interfaces */
        struct      ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define  IFF_UP          0x1   /* interface is up */
#define  IFF_BROADCAST   0x2   /* broadcast address valid */
#define  IFF_DEBUG       0x4   /* turn on debugging */
#define  IFF_ROUTE       0x8   /* routing entry installed */
#define  IFF_POINTOPOINT 0x10  /* interface is point-to-point link */
#define  IFF_NOTRAILERS  0x20  /* avoid use of trailers */
#define  IFF_RUNNING     0x40  /* resources allocated */
#define  IFF_NOARP       0x80  /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets,

the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct  ifuba {
        short       ifu_uban;               /* uba number */
        short       ifu_hlen;               /* local net header length */
        struct      uba_regs *ifu_uba;      /* uba regs, in vm */
        struct ifrw {
                    caddr_t   ifrw_addr;  /* virt addr of header */
                    int       ifrw_bdp;   /* unibus bdp */
                    int       ifrw_info;  /* value from ubaalloc */
                    int       ifrw_proto; /* map register prototype */
                    struct    pte *ifrw_mr;/* base of map registers */
        } ifu_r, ifu_w;
        struct      pte ifu_wmap[IF_MAXNUBAMR];/* base pages for output */
        short       ifu_xswapd;             /* mask of clusters swapped */
        short       ifu_flags;              /* used during uballoc's */
        struct      mbuf *ifu_xtofree;      /* pages being dma'd out */
};
```

The *if_uba* structure describes UNIBUS resources held by an interface. IF_NUBAMR map registers are held for datagram data, starting at *ifr_mr*. UNIBUS map register *ifr_mr*[-1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifr_bdp*. The prototype of the map registers for read and for write is saved in *ifr_proto*.

When write transfers are not full pages on page boundaries the data is just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is of a (1024 byte) page size and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifu_wmap* when the transfer completes.

When read transfers give whole pages of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers, all use the *ifuba* structure described above.

if_ubainit(ifu, uban, hlen, nmr);

if_ubainit allocates resources on UNIBUS adaptor *uban* and stores the resultant information in the *ifuba* structure pointed to by *ifu*. It is called only at boot time or after a UNIBUS reset. Two data paths (buffered or unbuffered, depending on the *ifu_flags* field) are allocated, one for reading and one for writing. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with resources already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization is successful, 0 otherwise.

m = if_rubaget(ifu, totlen, off0);

if_rubaget pulls read data off an interface. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When page sized units of data are present and are page-aligned, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages; thus avoiding any copying. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifu, m);

if_wubaput maps a chain of mbufs onto a network interface in preparation for output. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Any other mbufs which contained non page sized data portions are also copied to the I/O space. Pages mapped from a previous output operation (no longer needed) are unmapped and returned to the network page pool.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH      0    /* attach protocol */
#define PRU_DETACH      1    /* detach protocol */
#define PRU_BIND        2    /* bind socket to address */
#define PRU_LISTEN      3    /* listen for connection */
#define PRU_CONNECT     4    /* establish connection to peer */
#define PRU_ACCEPT      5    /* accept connection from peer */
#define PRU_DISCONNECT  6    /* disconnect from peer */
#define PRU_SHUTDOWN    7    /* won't send any more data */
#define PRU_RCVD        8    /* have taken data; more room now */
#define PRU_SEND        9    /* send this data */
#define PRU_ABORT       10   /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL     11   /* control operations on protocol */
#define PRU_SENSE       12   /* return status into m */
#define PRU_RCVOOB      13   /* retrieve out of band data */
#define PRU_SENDOOB     14   /* send out of band data */
#define PRU_SOCKADDR    15   /* fetch socket's address */
#define PRU_PEERADDR    16   /* fetch peer's address */
#define PRU_CONNECT2    17   /* connect two sockets */
/* begin for protocols internal use */
#define PRU_FASTTIMO    18   /* 200ms timeout */
#define PRU_SLOWTIMO    19   /* 500ms timeout */
#define PRU_PROTORCV    20   /* receive from below */
#define PRU_PROTOSEND   21   /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH
: When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH
: This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND
: When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN
: The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to

accept a connection.

PRU_CONNECT

The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT

Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT

Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN

This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD

This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND

Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL

The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE

The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics,

protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB
Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR
The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR
The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2
The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO
A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO
A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV
This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND
This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

```
error = ip_output(m, opt, ro, allowbroadcast);
int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m);
struct mbuf *m;
```

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, info);
int req; caddr_t info;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN          0    /* interface transition */
#define PRC_ROUTEDEAD       1    /* select new route if possible */
#define PRC_QUENCH          4    /* some said to slow down */
#define PRC_HOSTDEAD        6    /* normally from IMP */
#define PRC_HOSTUNREACH     7    /* ditto */
#define PRC_UNREACH_NET     8    /* no route to network */
#define PRC_UNREACH_HOST    9    /* no route to host */
#define PRC_UNREACH_PROTOCOL     10/* dst says bad protocol */
#define PRC_UNREACH_PORT    11   /* bad port # */
#define PRC_MSGSIZE         12   /* message size forced drop */
#define PRC_REDIRECT_NET    13   /* net routing redirect */
#define PRC_REDIRECT_HOST   14   /* host routing redirect */
#define PRC_TIMXCEED_INTRANS     17/* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS  18   /* lifetime expired on reass q */
#define PRC_PARAMPROB       19   /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a VAX software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long      rt_hash;            /* hash key for lookups */
        struct      sockaddr rt_dst;    /* destination net or host */
        struct      sockaddr rt_gateway;/* forwarding agent */
        short       rt_flags;           /* see below */
        short       rt_refcnt;          /* no. of references to structure */
        u_long      rt_use;             /* packets sent using route */
        struct      ifnet *rt_ifp;      /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is

normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct      rawcb *rcb_next;        /* doubly linked list */
        struct      rawcb *rcb_prev;
        struct      socket *rcb_socket;     /* back pointer to socket */
        struct      sockaddr rcb_faddr;     /* destination address */
        struct      sockaddr rcb_laddr;     /* socket's address */
        caddr_t     rcb_pcb;                /* protocol specific stuff */
        short       rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct      sockproto raw_proto;
        struct      sockaddr raw_dst;
        struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth

network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols perogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present to fill an entire page). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without a priori knowledge of the format (e.g. supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol*, places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion, and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short       protocol;       /* original protocol no. */
        short       length;         /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates a trailer encapsulation is being used. The protocol identifier also includes an indication of the number of data pages present (before the trailer protocol header). The trailer protocol header is initialized to contain the actual protocol and variable length header size, and appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture – General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project - Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection – Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.

[Joy82a] Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; *4.2BSD System Manual.* Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.

[Postel79] Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.

[Postel80a] Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.

[Postel80b] Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.

[Xerox81] Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.

SENDMAIL – An Internetwork Mail Router

Eric Allman†

Britton-Lee, Inc.
1919 Addison Street, Suite 105.
Berkeley, California 94704.

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

SENDMAIL – An Internetwork Mail Router

Eric Allman†

Britton-Lee, Inc.
1919 Addison Street, Suite 105.
Berkeley, California 94704.

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

Sendmail implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characterstics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical organization of the address space.

---

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

Sendmail is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

1. DESIGN GOALS

Design goals for *sendmail* include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formated addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

(5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

2. OVERVIEW

2.1. System Organization

Sendmail neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission[1]. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

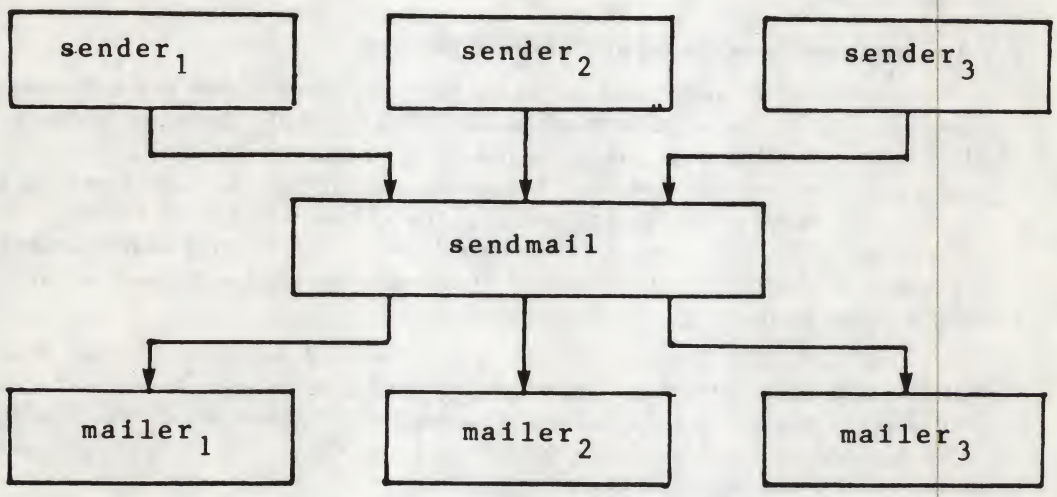


Figure 1 - Sendmail System Structure.

[1]except when mailing to a file, when *sendmail* does the delivery directly.

2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special format.

2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2BSD IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a sendmail process on another machine.

2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is deferred until delivery. Forwarding is also performed as the local addresses are verified.

Sendmail appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages deliverd to the same recipient, as might occur if a person is in two groups.

2.3.2. Message collection

Sendmail then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to sendmail. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

2.3.4. Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory[2].

2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling sendmail the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

3. USAGE AND IMPLEMENTATION

3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

(1) Anything in parentheses is thrown away (as a comment).

(2) Anything in angle brackets ("<>") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name <machine-address>

[2]Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message - the "return to sender" function is always handled in one of these two ways.

will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes (") quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently - for example, *user* and "*user*" are equivalent, but *user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[3].

3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

3.3. Aliasing, Forwarding, Inclusion

Sendmail reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"|/usr/local/newmail myname"

will use a different incoming mailer.

3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

---

[3]Disclaimer: Some special processing is done after rewriting local names; see below.

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

3.6. Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recomplied except

(1) To change operating systems (V6, V7/32V, 4BSD).

(2) To remove or insert the DBM (UNIX database) library.

(3) To change ARPANET reply codes.

(4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message if they don't exist in the incoming message. Certain headers are suppressed by some mailers.

3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

ucsfcgl!tef

might be mapped into:

tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

4. COMPARISON WITH OTHER MAILERS

4.1. Delivermail

Sendmail is an outgrowth of *delivermail.* The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail.* Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail.* For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code[4].

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in

---

[4]Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

[5]The MMDF equivalent of a *sendmail* "mailer."

MMDF they are split into two programs.

4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail*. When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples[6]. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

5. EVALUATIONS AND FUTURE PLANS

Sendmail is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

Sendmail has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

---

[6]This is similar to the NBS standard.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

ACKNOWLEDGEMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and BerkNet respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.

REFERENCES

[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25,* 4, April 82.

[Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual.* R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages.* RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System.* R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility – MMDF.* 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19,* 7. July 1976.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook.* NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard.* Report No. ICST/CBOS 80-2. October 1980.

[Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network.* RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b] Nowitz, D. A., *Uucp Implementation Description.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. October, 1978.

[Postel74] Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77] Postel, J., *Mail Protocol.* NIC 29588. In [Feinler78]. November 1977.

[Postel79a] Postel, J., *Internet Message Protocol.* RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b] Postel, J. B., *An Internetwork Message Structure.* In *Proceedings of the Sixth Data Communications Symposium,* IEEE. New York. November 1979.

[Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents.* RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.

[Postel82] Postel, J. B., *Simple Mail Transfer Protocol.* RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network.* University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol.* RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications.* RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83] *The UNIX Programmer's Manual, Seventh Edition,* Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.

On the Security of UNIX

Dennis M. Ritchie

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX† system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
        mkdir x
        cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each

† UNIX is a trademark of Bell Laboratories.

process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect *and*-ed with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login,* allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited. The encryption scheme used is not one of the strongest known, but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort

than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the crypt command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login:" on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su.* A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

Password Security: A Case History

Robert Morris

Ken Thompson

INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

---

† UNIX is a trademark of Bell Laboratories.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

THE FIRST SCHEME

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

ATTACKS ON THE FIRST APPROACH

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search.* Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It is takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length n that consist entirely of lower-case letters, the number of such passwords is 26^n. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than 95^n. (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length n chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

n	26 lower-case letters	36 lower-case letters and digits	62 alphanumeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	43 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.		
6	107 hrs.				

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing,

except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

IMPROVEMENTS TO THE FIRST APPROACH

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 (2^{12}). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

References

[1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* **17** (July 1974), pp. 365-375.

[2] *Proposed Federal Information Processing Data Encryption Standard.* Federal Register (40FR12134), March 17, 1975

[3] Wilkes, M. V. *Time-Sharing Computer Systems.* American Elsevier, New York, (1968).

[4] U. S. Patent Number 2,089,603.

PART 5: SUPPORTING DOCUMENTS

The remaining articles in this part are included for historical reasons. They provide background information, some of which you may find useful in installing or tuning your ULTRIX–32m system. However, much of the information in these articles is obsolete. Note in particular that first article on *uucp* does not refer to the current implementation. Check the *ULTRIX–32m Installation Guide* and the *ULTRIX–32m System Manager's Guide* for installation and maintenance procedures.

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

ABSTRACT

A network of over eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

August 18, 1978

† UNIX is a trademark of Bell Laboratories.

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

1. Purpose

The widespread use of the UNIX system[1] within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.[1,2] Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[3,4] Thus, systems typically join

our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.

- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp* or *uux* programs,

c) by a remote system.

Scan For Work

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

Ucode

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S	send a file,
R	receive a file,
C	copy complete,
H	hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.[5] Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.[6] There are distribution errors and delays, resulting in old software running at remote

sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1905-1929, 1978.
2. G. L. Chesson, "The Network UNIX System," *Operating Systems Review*, vol. 9, no. 5, pp. 60-66, 1975. Also in *Proc. 5th Symp. on Operating Systems Principles.*
3. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications*, p. 21F, June 1974. IEEE Cat. No. 74CH0859-9-CSCB.

4. A. G. Fraser, "A Virtual Channel Network," *Datamation*, pp. 51-56, February 1975.

5. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey, June 1976.

6. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass., 1975.

Uucp Implementation Description

D. A. Nowitz

Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.[1] This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execution files* are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp	This program creates work and gathers data files in the spool directory for the transmission of files.
uux	This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.
uucico	This program executes the work files for data transmission.
uuxqt	This program executes the execution files for UNIX command execution.

The secondary programs are:

uulog	This program updates the log file with new entries and reports on the status of uucp requests.
uuclean	This program removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

uucp [option] ... source ... destination

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

-d	Make directories when necessary for copying the file.
-c	Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

† UNIX is a trademark of Bell Laboratories.

1 M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.

-*gletter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

-m Send mail on completion of the work.

The following options are used primarily for debugging:

-r Queue the job but do not start *uucico* program.

-s*dir* Use directory *dir* for the spool directory.

-x*num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "*?*[]*". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

```
uucp *.c usg!/usr/dan
```

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the"usg" machine.

The source and/or destination names may also contain a *~user* prefix. This translates to the login directory on the specified system. For names with partial path-names, the current directory is prepended to the file name. File names with ../ are not permitted.

The command

```
uucp usg!~dan/*.h ~dan
```

will set up the transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote systems.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the -r option was specified).

Type 1

A *cp* command is used to do the work. The *–d* and the *–m* options are not honored in this case.

Type 2

A one line *work file* is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

[1] R

[2] The full path-name of the source or a ~user/path-name. The *~user* part will be expanded on the remote system.

[3] The full path-name of the destination file. If the *~user* notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A "-" followed by an option list. (Only the -m and -d options will appear in this list.)

Type 3

For each source file, a *work file* is created and the source file is copied into a *data file* in the spool directory. (A "-c" option on the *uucp* command will prevent the *data file* from being made.) In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ~user/file-name.

[4] The user's login name.

[5] A "-" followed by an option list.

[6] The name of the *data file* in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

Type 4 and Type 5

Uucp generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

2. Uux - UNIX To UNIX Execution

The *uux* command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

uux [-] [option] ... command-string

where the command-string is made up of one or more arguments. All special shell characters such as "<>|" must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) The "-" is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The options, essentially for debugging, are:

-r Don't start *uucico* or *uuxqt* after queuing the job;

-x*num* Num is the level of debugging output desired.

The command

pr abc | uux - usg!lpr

will set up the output of "pr abc" as standard input to an lpr command to be executed on system "usg".

Uux generates an *execute file* which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The *execute file* will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more

arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

U user system

where the *user* and *system* are the requester's login name and system.

Required File Line

F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

Standard Input Line

I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "-" option is used. If a standard input is not specified, "/dev/null" is used.

Standard Output Line

O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note - the use of ">>" is not implemented.)

Command Line

C command [arguments] ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

3. Uucico - Copy In, Copy Out

The *uucico* program will perform the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp, uux, uuxqt* or *uucico* programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the "shell" field in the "/etc/passwd" file for the "uucp" logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (-s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

-r1 Start the program in *MASTER* mode. This is used when *uucico* is started by a program or "cron" shell.

-s*sys* Do work only for system *sys*. If *-s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

-d*dir* Use directory *dir* for the spool directory.

-x*num* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

Scan For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where:

Type is an upper case letter, (*C* - copy command file, *D* - data file, *X* - execute file);

System-name is the remote system;

Grade is a character;

Number is a four digit, padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix "C."). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is;

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU*,

[6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where proto-list is a string of characters, each representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or *N* which means there is no common protocol.

Work Processing

The initial roles (*MASTER* or *SLAVE*) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the "-r1" uucico option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S send a file,

R receive a file,

C copy complete,

X execute a *uucp* command,

H hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, *XY*, *XN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the case of *CN*, the transferred file will be in the spool directory with a name beginning with "TM'.) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, an *HN* message is

sent and the programs switch roles. If no work exists, an *HY* response is sent.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a *–s* option was specified.

4. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the "Uux" section above.

Command Execution

The execution is accomplished by executing a *sh –c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

-s*sys*	Print entries where *sys* is the remote system name;
-u*user*	Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

-d*dir*	The directory to be scanned is *dir*.
-m	Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)
-n*hours*	Change the aging time from 72 hours to *hours* hours.
-p*pre*	Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
-x*num*	This is the level of debugging output desired.

7. Security

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require callback for certain login-ids. (See the "Files required for execution" section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specifed in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name.*

The four directories are:

lib	(/usr/src/cmd/uucp) This directory contains the source files for generating the *uucp* system.
program	(/usr/lib/uucp) This is the directory used for the executable system programs and the system files.
spool	(/usr/spool/uucp) This is the spool directory used during *uucp* execution.
xqtdir	(/usr/spool/uucp/.XQTDIR) This directory is used during execution of *execute files.*

The names given in parentheses above are the default values for the directories. The italicized named *lib, program, xqtdir,* and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

makefile modification

There are several *make* variable definitions which may need modification.

INSDIR This is the *program* directory (e.g. INSDIR=/usr/lib/uucp). This parameter is used if "make cp" is used after the programs are compiled.

IOCTL This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard "C" library; the statement "IOCTL=ioctl.o" is required in this case.

PKON The statement "PKON=pkon.o" is required if the packet driver is not in the kernel.

Compile the system The command

make

will compile the entire system. The command

make cp

will copy the commands to the to the appropriate directories.

The programs *uucp*, *uux*, and *uulog* should be put in "/usr/bin". The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

Files required for execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

line call-unit speed

where;

line is the device for the line (e.g. cul0),

call-unit is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number "0" in this field.),

speed is the line speed.

The line

cul0 cua0 300

would be set up for a system which had device cul0 wired to a call-unit cua0 for use at 300 baud.

L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

abb dial-seq

where;

abb is the abbreviation,

dial-seq is the dial sequence to call that location.

The line

py 165-

would be set up so that entry py7777 would send 165-7777 to the dial-unit.

LOGIN/SYSTEM NAMES

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique *system name* which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

USERFILE

This file contains user accessibility information. It specifies four types of constraint;

[1] which files can be accessed by a normal user of the local machine,

[2] which files can be accessed from a remote computer,

[3] which login name is used by a particular remote computer,

[4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

login,sys [c] path-name [path-name] ...

where;

login	is the login name for a user or the remote computer,
sys	is the system name for a remote computer,
c	is the optional *call-back required* flag,
path-name	is a path-name prefix that is acceptable for *user*.

The constraints are implemented as follows.

[1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

[2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a *null* system name is used.

[3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.

[4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

u,m /usr/xyz

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

dan, /usr/dan

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan".

The lines

```
u,m /usr/xyz  /usr/spool
u,  /usr/spool
```

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool".

The lines

```
root,  /
,  /usr
```

allows any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name

The name of the remote system.

time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800-1730).

The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800-1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

speed

This is the line speed for the call (e.g. 300).

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995-9980).

For the hardwired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in the format

expect send [expect send] ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect[-send-expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file would be

```
sys Any ACU 300  mh7654 login uucp ssword: word
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

SQFILE – sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

TM – temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

```
uuclean -pTM
```

will remove all *TM* files older than three days.

LOG – log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

```
uulog
```

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new LOG entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

STST – system status files

These files are created in the spool directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when to machines are conversing. The form of the file name is

STST.sys

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

LCK – lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

LCK..str

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a "shell" file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

program/uulog
program/uucico -r1

Note that the "-r1" option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

program/uuclean -pTM -pC. -pD.
program/uuclean -pST -pLCK -n12

can be used. Note the "-n12" option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the "-n" option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like

cp *spool*/LOGFILE *spool*/o.LOGFILE
rm *spool*/LOGFILE

can be used.

Login Entry

One or more logins should be set up for *uucp*. Each of the "/etc/passwd" entries should have the "*program*/uucico" as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of uucp (*uucico*) only.

File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico* and *uuxqt* should be owned by the *uucp* login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

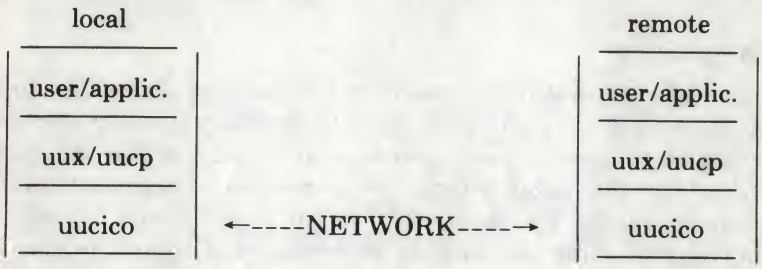
UUCP Installation and Administration

1. Introduction

The purpose of this document is to describe the installation and administration of the current version of *uucp*. The current release is an enhancement of the UNIX† version 7 *uucp* software. Major enhancements include improved performance, security, and numerous bug fixes. A quick reference guide is provided in the appendix.

1.1. Brief Overview of the UUCP System

The *uucp* system consists of three program levels as follows:



At the highest logical level is the user or application program. This level will use *uux* or *uucp* to initiate requests for file transfers and/or remote job execution. *uux* is the remote execution program and *uucp* is the program that spools user requests for file transfers. Both programs convert higher level requests into the format required by the file transfer daemon *uucico*. Each request is queued in the appropriate spool directory.

uucico is the program which performs the file transfer over the network. Before the transfer takes place though, *uucico* must go through several stages. First *uucico* must call up the destination system and log in. After a successful login, the handshaking stage takes place between the destination *uucico* daemon process and the local daemon. At this stage the daemons will determine if the opposing systems have permission to use the local resources at each machine. If the handshake succeeds the daemons will then select which protocol will be used to ensure that raw data is reliably transmitted over the network. Each daemon will then use the corresponding packet driver software to send and receive raw data. At this point the file transfer process begins. The local daemon will search the spool directories for job requests, build a list of files to transfer, and then start transmitting the files. A file transfer protocol is used to ensure that each file is successfully transmitted only once or to notify the user that the file could not be transmitted with the reason for failure. After all of the files have been transmitted the destination site will begin to transfer files back to the local system. When both systems have no more work to be done, the connection through the network is broken and the conversation is complete.

Throughout the conversation temporary files are created and removed. Lock files are created in the top level spool directory (usually /usr/spool/uucp) that correspond to the remote system (LCK..sys) and the hardware device used for communication (for example, LCK..cua0). Status

This document is partially based on work done by D. A. Nowitz and M. E. Lesk.
† UNIX is a trademark of Bell Laboratories.

files (STST. files) are updated that keep track of accessability of each system. Many other temporary files are created, all of which should be gone by the time the conversation is complete.

The *uucp* system also includes the remote execution programs: *uux* and *uuxqt*. *uux/uuxqt* will execute commands on a specified remote system. *uux* spools the command and associated data into the appropriate spool directories. *uucico* will then transfer the files to the remote system. At the remote system *uuxqt* will start when these execution files arrive. *uuxqt* will scan through the execution spool directories (X.) searching for commands to execute. If a command has arrived along with the data files needed by the command, *uuxqt* will try to execute the command. *uuxqt* also creates LCK files. LCK files are created for the type of command it is looking for (for example, LCK.XQTrmail or LCK.XQT for all commands) and a LCK file for the command *uuxqt* is currently working on (for example, LCK.X.decvaxX0123).

In the event that some temporary files do not get removed, shell scripts are run periodically to clean up the residue. In some instances manual intervention may be necessary. This document will provide the tools needed by the system manager to maintain the system and resolve problems when they arise.

1.2. Enhancements

1.2.1. Improved Spooling

The most significant enhancement is the improved file spooling algorithm. In the original version all work related files were placed in the SPOOL directory (usually /usr/spool/uucp). The SPOOL directory has now been split out into separate subdirectories including the option of per system subdirectories. This subdirectory scheme prevents a huge buildup of data files from slowing down the *uucp* transfer process (*uucico*). The per-system subdirectories prevent the failure of one remote system from effecting the performance of communications with other systems.

In addition to the subdirectory enhancements the methods for scanning the spool directories has been improved. All of these changes have significantly improved *uucp's* ability to handle greater loads.

1.2.2. Improved Security

uucp will now check to see if the remote system has permission to login to the local system. The action taken by *uucp* (accept/reject) depends on the contents of the USERFILE and the existence of the file /usr/lib/uucp/INSECURE. Also, the USERFILE now provides remote execution security. A more complete discussion of the USERFILE and *uucp* security is presented later.

1.2.3. Miscellaneous Enhancements and Bug Fixes

In addition to the above enhancements numerous bug fixes have been made as well as some internal improvements. The *uustat* program has been installed and improved. *uustat* provides job status and control to *uucp* users. This version also supports several modems/auto call units (acu) including: the DF03/DF02, Bell 212/801, Hayes Smartmodems, and Ventel MD212.

1.3. Caveat

uucp has been improved over earlier versions. However, this does not imply that *uucp* is now self administering. A watchful eye is still needed. Given the loads that can be generated by the USENET or other high demand programs, a failure of one or more sites can cause sizable backlogs of files. These anomalous situations must still be handled manually. At least in this present version something can be done to rectify the problem (other than trashing files). The topic of what to do when something goes wrong will be discussed later.

2. Installation

uucp installation requires a three-step process. However, before you can install *uucp*, you must select the appropriate hardware.

- Install the appropriate hardware (See Subsection 2.1)
- Set up the special files corresponding to the hardware (See Subsection 2.2)
- Install the minimum set of *uucp* directories (See Subsection 2.3.8.1)
- Install *uucp* commands (if necessary)
- Update Administrative files: USERFILE, L.sys, L-devices, L.cmds

2.1. Installing Supported Hardware

uucp currently operates with the following hardware (only software written for DIGITAL hardware is supported by DEC):

- Bell System 801 type Auto Call Unit (ACU) with with a DN-11 line card and 212 type data set
- DEC DF02 or DF03 Auto Call Units with modem
- Direct connect with a null modem cable such as a BC03-M
- Direct connect with a modem link
- Hayes Smartmodem1200
- Ventel MD212 models
- Racal-Vadic 3450

A brief overview of hardware installation procedures follows. See the User's Guide for specific devices, or the System Installation Guide for more detailed information.

2.1.1. Installing Bell System Hardware

The 801 ACU has four option switches that you must set as follows:

(0=open 1=closed)

S1 = 1000[1]
S2 = 0101
S3 = 11010
S4 = 11[00]

Set the 212 data set as follows:

S1 = [0]001
S2 = 110001100
S3 = 11110000
S5 = 00

Press the high speed button in for 1200 baud. Leave the button out for 300 baud.

2.1.2. Installing DEC ACUs

Connect the DF02-AC or DF03-AC auto call modem to a port on the local system using a straight-through cable. Connect the ACU to the phone line by following the instructions in the *DF02-AC or DF03-AC Modem User's Guide*. Set the ACU communications bit rate, see switch options on the DF02/03-AC ACU in the modem user's guide. If the ACU is connected to a

DL11 interface, then the ACU communications bit rate must match the DL11 speed. If the operating system is ULTRIX-11 then the ACU dialing rate (not the modem speed) must be set to 300 BPS. Otherwise 1200 BPS can be used.

2.1.3. Installing Other Brands of Modems

Consult the manufacturer's installation guide for the specific modem to use. One hint is to configure the modem such that commands typed to the modem are not echoed back by the modem.

2.1.4. Installing Direct Connects

To install a hardwired direct link, connect a null modem cable from a port on the local system to a port on the remote system. For a direct link with a modem, connect the modem to a port on the local system to a port on the remote system with a straight-through cable. Follow the modem installation instructions to setup the direct modem link. A direct connect should be able to run at speeds as high as 9600 BPS.

2.2. Creating Special Device Files

After connecting the hardware, you create the corresponding special files. A special file must exist for both the ACU and modem line. You need only one special file for modems with integral dialers, such as DF03s, because the ACU and modem appear as one unit to the software. The following naming conventions are normally used:

cua#	for an ACU
cul#	for the associated modem line
ttyab	for a direct connect between systems a and b
tty#	another common convention for a direct connect

The modes of special files files should be 0666. Lines that are used to initiate conversations must not have a getty(2) process running on them at the local system. Also, when the system is configured, the tty lines must be set up correctly. Outgoing lines should ignore the carrier. Only incoming *modem* lines will wait for the carrier.

2.3. Installation of UUCP Software

These files and directories must be set up by the system *uucp* manager before any programs can be run:

LIB/USERFILE	defines *uucp* security
/etc/passwd	password file
LIB/L.sys	information needed to connect to a system
LIB/L-devices	devices used to connect to remote systems
LIB/L-dialcodes	dial code abbreviations
LIB/L.cmds	allowable remote execution commands
LIB/Makefile	*uucp* configuration and compilation
/etc/uucpname	local system name
spool directories	directories for depositing spooled files and temporary work files.

where,

LIB = /usr/lib/uucp

After the previous files have been created you can install the *uucp* commands using Makefile. NOTE: If the *uucp* software has already been installed (binary only systems) then the Makefile need not be modified.

2.3.1. USERFILE and UUCP Security

There are two mechanisms available to protect systems:

1) Normal UNIX system file access modes

2) The USERFILE

All UNIX system users should be well aware of method one. Each user should have their files properly protected. In its most insecure state *uucp* can grab files that are readable by *uucp* (for example, mode xx4). It can also overwrite files that are writable by *uucp*. Ultimately, the users are responsible for the security of their own files.

The USERFILE is the first line of defense from remote users. It provides three flavors of security:

- File access permission for remote and local users.
- Log in security.
- Remote execution permissions

Each line in the file has the format:

login,[sys] [X#] [c] path-name [path-name] ....

In this format

login	is the login name for a remote system or local user.
sys	is the name of the remote system and is optional (read following discussion)
X#	is the level of execution assigned to *sys* and is optional, except in the default entries, (See Subsection 2.3.1.2 for explanation of defaults).
c	is the optional call-back required flag
path-name	is a path-name prefix that *sys* can access

2.3.1.1. File Access Security with the USERFILE

The domain of accessibility of a remote system is restricted by the USERFILE. An entry should exist for each system that defines which paths the system can access. If no entries exist for a particular system the default entries will be used, (See Subsection 2.3.1.2 for explanation of defaults).

NOTE: The USERFILE must be set up with default entries that use this format:

```
remote, X#    /some_path_name_for_remote_systems
local, X#     /some_path_name_for_local_users
```

The letter X is a keyword used by *uucp* to identify the set of commands that a remote system may execute on the local system. In the following examples the X field is included for accuracy only; it is discussed in the section covering remote execution security. The words local and remote are also key words used by *uucp*.

remote - this entry is the default entry for remote systems that do not have an explicit entry in the USERFILE. Do not be too liberal with this entry. A typical path allowed for remote

users would be:

```
remote, X1      /usr/spool/uucppublic
```

/usr/spool/uucppublic is a well known public repository. Users should not leave important files in that area.

local - this entry is the default entry for local users. Usually the normal UNIX system file access modes are all the security that is imposed on local users. Therefore, the typical default entry for local users is:

```
local, X9  /
```

Caution: The default entries must be supplied, otherwise the *uucp* software will fail.

In addition to the default entries, per-system entries may also be supplied. This provides the flexibility to give trustworthy systems less restrictive access to the local system. The following examples illustrate the alternatives:

Example 1:

```
remote, X1      /usr/spool/uucppublic
local, X9       /
max,systemX     /usr/sources
```

This example allows the remote system systemX, which has logged into the local system with the login name max, to access anything that has the prefix /usr/sources. All other systems can access the public directories only.

Example 2:

```
remote, X1      /usr/spool/uucppublic
local, X9       /
max, /usr       /usr/src/share
```

This example shows that ANY system or user that has successfully logged into the local system with login name max to access anything in or below /usr or /usr/src/share. Note that several systems could log in with this same login name so care should be take to restrict access rights appropriately. All other systems can access the public directories only.

2.3.1.2. LOGIN Security with the USERFILE

uucp tries to ensure that only legitimate systems log in to the local system. When a remote system logs in, it passes its node name to the local system. *uucp* crosschecks the node name and login name against the USERFILE. If an entry exists for that node name and the login name does not match the one specified in the USERFILE, the connection attempt will be aborted. The following example illustrates this situation:

example USERFILE:

```
remote, X1          /usr/spool/uucppublic
local, X9           /
max,systemY c       /usr
max,systemZ         /sys
blimpy,systemQ      /
nuucp,              /usr/spool/uucppublic
```

Scenario 1:

The competition across the street has unscrupulously obtained the login name blimpy and

password for your *uucp* system. They successfully connect to the local *uucp* system. The local *uucp* then checks for a USERFILE entry for blimpy. It finds the entry and knows that only systemQ can connect with the login name blimpy. Since the nodename of the remote system is syszero, the connection attempt fails.

Scenario 2:

In this case the same competition stole the login entry for *nuucp*. Since no system name is provided in the USERFILE for the *nuucp* entry the connection attempt will succeed.

Scenario 3: (Default login security)

The situation may occur where a system logs in successfully and there is NO entry in the USERFILE that corresponds to either the login name or the remote node name. *uucp* handles this situation as follows:

> IF the file /usr/lib/uucp/INSECURE exists, the connection request is accepted. If /usr/lib/uucp/INSECURE does not exist, the connection request is rejected.

This option is provided so that the system manager need not supply an entry for every system that can log in. The default (remote) entry is used for systems that do not have an entry in the USERFILE. An example of this is where the system manager only wants to worry about two *uucp* passwords: one login for trustworthy systems and another login for the other systems (for USENET perhaps).

Here is a sample USERFILE.

```
remote, X1     /usr/spool/uucppublic
local, X9      /
safeuucp,      /
unsafeuucp,    /usr/spool/uucppublic
```

Since no system names are specified, connection attempts that use the logins *safeuucp* or *unsafeuucp* succeed. Attempts to connect with other logins succeed only if /usr/lib/uucp/INSECURE exists and then will only receive the default path security.

Scenario 4: (call back option)

If the system manager believes that it is possible to forge remote node names, the call back option can be used. In the USERFILE example, if a successful login is made from a node that claims to be systemY, *uucp* will reject the request and then try to call the real systemY. This is the most secure form of USERFILE entry. Note that if both the destination and local system use the call back option, an infinite loop of call backs can occur. Make sure this does not happen.

2.3.1.3. Remote Execution Security

A new security enhancement allows the system manager to restrict remote execution to specified systems only. Also, systems that are allowed to execute commands on the local system can be limited to a subset of allowable commands.

The X# field of a USERFILE entry is used to provide levels of security. The # can range from zero to nine where zero is the most secure level and nine is the least secure level. When the execution daemon (*uuxqt*) processes a remote execution request, it uses this algorithm:

> *uuxqt* obtains the security level from the USERFILE that corresponds to the remote system making the request. The command to be executed also has a level number. (See Subsection 2.3.6 for information about the L.cmds file). If the level

associated with the remote system is greater than or equal to the number associated with the command, then *uuxqt* grants permission to execute that command. Otherwise the remote execution request fails.

NOTE: You must specify the execute field in the default USERFILE entries; otherwise, the *uucp* software will fail. All other USERFILE entries need not have the execute level specified. For entries that don't have an execution level listed, then the default execution level will be used for that system. The default execution level for remote machines is obtained from the remote USERFILE entry. The local USERFILE entry provides the execution level to local users.

NOTE: Execution levels are provided on a machine-name basis only, not to particular users. Thus, you cannot use a USERFILE entry that specifies a login name but no system name to determine the execution level of a system that logs in with this entry. Instead, machines that do not have their own USERFILE entry will use the default *remote* USERFILE entry.

The example below illustrates remote execution security:

Assume the following USERFILE and L.cmds file.

USERFILE:

```
remote,  X0            /usr/spool/uucppublic
local,  X9             /
maxuucp,sysmax  X3     /usr
xuucp,systemx  X1      /usr/spool/uucppublic
yuucp,systemy  X1      /usr/spool/uucppublic
zuucp,systemz          /usr/spool/uucppublic
nuucp,                 /usr/spool/uucppublic/stockroom
ruucp,  X5             /usr/spool/uucppublic/stockroom
```

L.cmds file:

```
rmail  X1
rnews  X1
uusend  X2
```

Explanation: The default *remote* entry prevents any system that does not have its own USERFILE entry, from executing any of the commands in L.cmds. They can send/receive files from the public directories. The systems *sysmax, systemx*, and *systemy* can execute rmail and rnews. *sysmax* is the only remote system that can execute uusend. Any system that logs in as *nuucp* is provided the default execution level (which in this case is zero) and thus can not execute any command. The important point to notice is that these latter systems (those that log on as *nuucp*) are not provided the default path security. Therefore the systems that log on as *nuucp* can only send/receive files to and from

/usr/spool/uucppublic/stockroom.

Any system that logs on as *ruucp* has the same execute permissions as those that log on as *nuucp*. Since no system name is provided in the *ruucp* USERFILE entry, the system ignores the X field and uses the default execution level instead.

Local users can access all of the commands in L.cmds.

2.3.2. Setting Up the Password File

Any system that wishes to connect to the local system must have an entry in /etc/passwd. System managers must agree on the login and password to be used. A typical password entry uses this format:

login:passwd:uid:gid:useful_info:/usr/spool/uucppublic:/usr/lib/uucp/uucico

The most important detail to note is that the last field is the path of the transfer daemon (*uucico*). When a remote system successfully logs in, *uucico* is run in the slave mode, and the dialog between the peer *uucicos* begins. It is good practice to have a USERFILE entry for every /etc/passwd entry, (See Subsection 2.3.1.2 on USERFILE security).

In addition to the latter /etc/passwd entries, an administrative login must be created for user: *uucp*.

2.3.3. The L.sys File

The L.sys file contains entries for each remote system that the local system can call. More than one line may be present for a particular system. In this case, the additional lines represent alternative communication paths that will be tried in sequential order. The format of each entry is:

system_name time device class phone login_sequence

Separate each field by blanks or tabs. Here are descriptions of the fields.

system name

The name of the remote system.

time

time is a string that indicates the days-of-week and times-of-day when the system can be called, for example, MoTuTh0800-1740.

The day portion may be a list containing:

Su Mo Tu We Th Fr Sa

Day may also be *Wk* for any week-day or *Any* for any day.

Indicate hours in a range, for example, 0800-1230. If you do not specify a time portion, any time of day is assumed to be allowed for the call. Note that a time range that spans 0000 is permitted.

For example, 0800-0600 means all times are allowed other than times between 6 and 8 am. Multiple date specifications, separated by | are allowed.

For example, Any0100-0600|Sa|Su means that the system can be called any day between 1 am and 6 am or any time on Saturday or Sunday.

An optional subfield is available to indicate the minimum time (minutes) before retrying a failed connection. A failed connection attempt is a log in failure as opposed to a dialing (connection) failure. The subfield separator is a comma (,). For example, *Any,9* means call any time but wait at least 9 minutes after a failure has occurred.

device

The device is either the *ACU* or the hard-wired device used for the call. For the hard-wired device, use the last part of the special file name, for example, tty2.

class

Class is the line speed for the call, for example, 1200. The exception is when the **C** library routine dialout is available, in which case this is the dialout class.

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation should be one that appears in the *L-dialcodes* file, for example, ct5900, nh6511, or an unabbreviated phone number. For the hard-wired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in this format:

expect[-sendspecial-expect] send ...

Expect is the string expected to be read when logging into the remote system, and *send* is the string to be sent when the *expect* string is received. If *expect* is not received, the subfield can be set up so that special characters (*sendspecial*) can be transmitted to the remote site. After the special characters are transmitted the *expect* following *sendspecial* is the next expected string. Two special characters which can be sent when *expect* is not received are *EOT* and *BREAK*. EOT will send an EOT character, and the string BREAK will send three break sequences (the break is simulated by using line speed changes and null characters). A number from one to nine may follow the BREAK. For example, BREAK1 will send 1 break. Note that after every send string a \r (carriage return) is sent except as noted below. If *sendspecial* is two consecutive dashes (--), a carriage return will be sent. In some instances it is necessary to send characters to the remote system before expecting something to arrive, for example, some systems want a carriage return before issuing a login prompt. A sequence of two double quotes ("") is useful in this regard. If "" is used as the expect string then nothing is expected and the following send string is transmitted. For example, "" \r\c will expect nothing, then send a carriage return. \c means the default \r should not be sent. If we did not put the \c here, two carriage returns would be transmitted.

These examples illustrate alternative expect-send sequences.

EXAMPLE 1:

login:--login xuucp ssword: foobaz

Explanation: *login:* is expected. When it is received, *xuucp* is sent. Now the word *ssword* is expected (the first letter of password varies from system to system so it is safer to look for the tail end, for example, *ssword*). When *ssword* is received *foobaz* is sent. If the login is successful, the conversation between the peer transfer processes (*uucico*) begins. If the login fails the connection attempt fails.

EXAMPLE 2:

login:--login: xuucp ssword: foobaz

Explanation: expect *login:*, if not received then send a carriage return and expect *login:* again. If receive *login* send *xuucp* and so on and so on.

EXAMPLE 3:

login:-BREAK1-login: xuucp ssword: foobaz

Explanation: expect *login:*, if not received send one break sequence (to change the baud rate of the remote *getty* process) and expect *login:*. Proceed as in the previous examples.

Other special characters can be used as part of the send sequence when talking to systems that are either slow or that need to be placed in a sane state before the true login sequence can begin. Here are the special characters and their meanings:

PAUSE[#]	pause # of seconds (or five seconds by default).
\d	pause one second before sending next character.
\s	send a blank character.
\r	send a carriage return.
\c	do not send a \r at the end of send string.
\b	send a break character.
\###	send the character represented by octal number ### (for example, \05 is control-e)
P_ZERO	change parity from even (default) to zero.
P_EVEN	change parity to even.
P_ODD	change parity to odd.
P_ONE	change parity to one parity.

Here are examples that illustrate the use of these special characters.

EXAMPLE 4:

"" @ login: xuucp ssword: foobaz

Explanation: expect nothing, send an @ character (line kill), send a carriage return, continue normal login sequence. The @ character is often useful for clearing out line noise before starting to log on. The default *line kill* character varies from system to system.

EXAMPLE 5:

"" P_ZERO "" \d\b "" \005\c login: xuucp ssword: foobaz

Explanation: expect nothing, change output parity to zero parity, expect nothing, delay one second then send a break sequence, expect nothing, send the escape character \005 without the default carriage return, continue with normal login sequence.

Putting all the fields together the following examples illustrate complete L.sys entries:

sys1 Any ACU 1200 wy7777 "" \r\d ogin:-EOT-ogin: Ufoobaz ssword: secret

sys1 Any ttyae 9600 tty32 "" \r\d ogin:-EOT-ogin: Ufoobaz ssword: secret

sys2 Any ACU 300 456-1234 "" \r ogin:-EOT-ogin:-EOT-ogin: Usys1 ssword: testing

sys3 Any,10 ACU 1200 8=123456789 login:-\r\c-login:-\r\c-login:-BREAK-login:-EOT-login: Uconn ssword: huskies

sys4 Any0000-0700 ACU 1200 8=987654321 login:-BREAK-login:-BREAK-login:-BREAK-login: Usys2 ssword: foobar

If the remote system uses nonstandard hardware the L.sys entry can be become complex. To connect to some systems you may have to alter the L.sys entries until a successful combination is found. Section 3. contains information which is useful for debugging L.sys entries.

NOTE: The L.sys file must also contain an entry for the name of any system that calls you, but you do not call them. The form of entry is abbreviated to system name and one word in place of the time entry, such as *never* or *incoming*.

2.3.4. The L-devices File

This file contains information about call units and direct connections. It is used to map specifiers in the L.sys file to specific devices. The format for each entry is:

type line call-unit speed brand

In this format:

type is a device type such as ACU or DIR. DIR indicates that this is a direct connect, hard-wired line.

line is the device for the modem line or hard-wired line as named in */dev*, for example, cul0 or ttyab. If the *type* is a DF02 or a DF03, (or any modem with an integral Auto Call Unit), this field will contain the automatic call device, for example, acu0. The special device files are assumed to be in the */dev* directory.

call-unit
is the automatic call unit associated with *line*, for example, cua0. Hardwired lines should place the device for the line in this field, for example, ttyab.

speed
is the line speed.

brand
is the brand name of the modem/acu. Here are the currently acceptable brands: dn11 (for Bell 801), DF02 or DF03 (for DEC modems), Ventel, Hayes, and Vadic. For direct connections, place the word direct should be placed in this field.

Here are typical *L-devices* entries:

ACU cua0 cua0 300 DF02
ACU cua1 cua1 1200 DF03
ACU cul3 cua3 1200 dn11
DIR ttyab ttyab 9600 direct

2.3.5. L-dialcodes

This file contains the dial-code abbreviations used in the *L.sys* file, for example, nh, boston. The entry format is:

abb dial-seq

In this format:

abb is the abbreviation as used in the L.sys file.
dial-seq is the dial sequence to call that location.

The entry **nh 603** would force any L.sys entry that used the prefix *nh* in the *phone* field, to send *603* to the dial unit before the rest of phone number is dialed.

2.3.6. L.cmds File

The L.cmds file is a list of commands that a remote system can execute via **UUX**. The format of the file follows:

command X#

command is a unix command or application program. *X#* is the execution level associated with *command*. The # can range from zero to nine. If the X field is not present then nine is provided as the default level. If X is present but a level number is not specified then zero is assumed (therefore, any system can execute this command). Care should be taken to limit the number and type of allowable commands, (See Subsection 2.3.1.3 on remote execution security). A typical list of commands would be:

rmail X1
rnews X1
uusend X1

To specify the path that *uuxqt* will use to search for commands, add the following line to L.sys (anywhere in L.sys will do):

PATH=path1:path2:path3:path4: (and so on)

For example,

PATH=/bin:/usr/bin:/usr/ucb

will tell *uuxqt* to first try /bin for a command. If the command is not in /bin, then try /usr/bin and then /usr/ucb. If no PATH entry is supplied, the default: PATH=/bin:/usr/bin will be used.

2.3.7. Makefile

The Makefile may need to be modified to reflect the desired *uucp* configuration options.

Add or delete -D fields in the CFLAGS line to change the configuration. The following options are available:

NDIR

If the operating system is a BSD version that is older than 4.1c, then the new directory system calls will have to be simulated. To simulate calls, create and install a directory library and header file (See Subsection 2.3.9 on installing *uucp* commands). By defining NDIR, the new header file (ndir.h) will be used instead of the standard directory header file (dir.h).

V7M11

Use V7M11 if the operating system in use is ULTRIX-11. NDIR is automatically defined by the software if V7M11 is defined.

UNAME

Use UNAME if the gethostname(2) system call is to be used to determine the local host name.

UUNAME

Use UUNAME if the file /etc/uucpname is to be used to determine the local host name. If this option is chosen the system manager must modify /etc/uucpname to reflect the correct local host name.

NOTE: *uucp* requires that node names contain seven characters at most. Longer names will be truncated to seven characters.

UUSTAT

Use UUSTAT if the *uustat* is to be turned on.

NOTE: If *uustat* is turned on, two files must be created before the *uucp* commands can be used. They are:

/usr/lib/uucp/L_stat #machine status file

/usr/lib/uucp/R_stat #uucp request status file

L_stat and R_stat must be owned and readable by *uucp*.

NOTE: If L_stat and R_stat exist from a previous version of *uucp*, THEY MUST BE CLEARED OUT.

The new version of *uustat* uses a different format (binary as opposed to ASCII) for these files. If they are not cleared out, the *uucp* commands will eventually fail. To clear out old L_stat and R_stat files enter the following commands:

```
cp /dev/null /usr/lib/uucp/L_stat
cp /dev/null /usr/lib/uucp/R_stat
```

The default CFLAGS line in the Makefile is:

CFLAGS=-O -DUUSTAT -DUNAME

The *uustat* program is turned on and the gethostname(2) system call is to be used. For ULTRIX-11 systems the default CFLAGS line is:

CFLAGS=-O -DUUSTAT -DUUNAME -DV7M11

The *uustat* program is turned on, all ULTRIX-11 specific code will be compiled, and /etc/uucpname will be used to obtain the local node name. A complete list of make command options is given in the beginning of the Makefile, for example, make *install*, make *uucp*, ..., (See Subsection 2.3.9 for instructions on how to use the Makefile for installing the *uucp* commands).

2.3.8. Spool Subdirectories

Subdirectories must be created for various *uucp* files as follows:

SPOOL/TM.	/* temporary files */
SPOOL/STST.	/* connection status files */
SPOOL/sys	/* system subdirectories */

The SPOOL is normally /usr/spool/uucp.

In addition, the *sys* subdirectory is further subdivided into directory names that correspond to specific remote systems and a default directory (DEFAULT). The minimum configuration requires that the SPOOL/sys/DEFAULT directory be created. Per system subdirectories must be created manually. To facilitate the latter operation the command:

/usr/lib/uucp/uumkspool *systemname*

will create the subdirectories for the specified system. Each system directory will contain the following subdirectories:

D.localname	/* outgoing data files */
D.localnameX	/* outgoing remote execution files */
D.	/* incoming data files */
C.	/* work files */
X.	/* incoming remote execution request files */

If a large amount of traffic is expected between specific systems, then you should create subdirectories for those systems. If, after a period of time, it becomes necessary to create new system directories, files will have to be moved out of the DEFAULT directory to the new system spool directory. The command */usr/lib/uucp/uurespool* will ease this process. *uurespool* is described below.

2.3.8.1. Installing Subdirectories on New Systems

The Makefile includes shell scripts to create the needed subdirectories.

i. Use the following shell script to create the minimum set of directories:

Enter the directory that contains the *uucp* Makefile.
Then type:

```
cd /usr/lib/uucp
su root
make mkdirs
```

NOTE: some directories may already exist.

If the above does not succeed try the following:

```
su root
cd /usr/spool/uucp/
mkdir TM. STST. sys
chown uucp TM. STST. sys
chmod 0755 TM. STST. sys
cd /usr/spool/uucp/sys
mkdir DEFAULT
chown uucp DEFAULT
chmod 0755 DEFAULT
cd DEFAULT
for i in C. D.localname D.localnameX D. X.
do
        mkdir $i
        chown uucp $i
        chmod 0755 $i
done
```

In this entry *localname* is the *uucp* node name of the local system.

ii. To create per-system subdirectories, type:

```
/usr/lib/uucp/uumkspool sys1 sys2 ...
```

where sys1 sys2 ... are the names of the system subdirectories

All of the required subdirectories should now exist.

2.3.8.2. Installing Subdirectories on Old Systems

If subdirectories are to be installed on a system that has been running with a different subdirectory scheme, follow these steps:

i. Follow steps i.-ii. in the previous section.

ii. It is now necessary to move files from the old directory to the new subdirectories.

Use the command */usr/lib/uucp/uurespool* to move old spool files to new spool directories. The *-t#* option allows you to specify the type of spool that was used prior to installing the new *uucp* system.

will be 1, 2, 3, or 4 as follows:

1 - if all spool files are located in /usr/spool/uucp. This is the format of the old *uucp*s. There are no subdirectories.

2 - if the spool directory has been split out into several subdirectories: D.local D.localX D. X. C.

3 - if the spool directory has been split as in 2 with the addition of C./OTHERS and STST.

4 - if a new system directory has been created and the spool files have to be moved from DEFAULT to the new system directory.

Execute the following command line:

/usr/lib/uucp/uurespool -t#

2.3.8.3. Adding Additional Per System Subdirectories at a Later Time

If additional per system subdirectories are to be added at a later date, the new directory needs to be created and spooled files moved from DEFAULT to the new directory. Run the following commands:

```
/usr/lib/uucp/uumkspool sys1 sys2 sys3 ....
/usr/lib/uucp/uurespool -t4
```

2.3.9. Installing UUCP Commands

If your site has object code only, ignore this section. Binary only sites have to install directories and administrative files only.

Follow these steps to install the *uucp* executables:

i. This step is required by systems that do not have the new directory system calls, (for example, versions older than 4.2BSD, and ULTRIX-11). Execute the following shell commands:

```
cd /usr/src/usr.lib/uucp      (or wherever you've stored theuucp sources)
cd ndir
make -f Makefile.V7M11 install
cd ..
```

ii. For BSD versions older than 4.2, Makefile must be modified to include:

LIBNDIR=/usr/lib/libndir.a

Also, the CFLAGS line must include NDIR and UUNAME (See Subsection 2.3.7 for description of Makefile).

In all systems, to save the old commands and install new commands, type:

make save

To install new commands only, type:

make install

You are now finished installing *uucp*.

3. UUCP Administration and Maintenance

As stated in the introduction, *uucp* is not self administering. If the system is to be used extensively, then an assigned system manager does the daily monitoring. To some extent *uucp* cleans up after itself using **crontab** shell scripts.

If the system uses the supported hardware, most problems should either be administrative (for example, files not set up correctly, wrong phone numbers) or hardware failures. Problems with the software may still exist but they should be much more rare. The discussion below should help determine the source of the problem.

3.1. Administrative Files

These administrative files contain *uucp* statistics to use for diagnosing problems:

/usr/spool/uucp/LOGFILE

/usr/spool/uucp/ERRLOG

/usr/spool/uucp/SYSLOG

3.1.1. LOGFILE

The LOGFILE contains information logged by a transfer process (*uucico*) concerning a conversation with a remote site. If a problem occurs during the connection stage, it noted here. The entry format is:

login_name (time of transaction-process_number) message

Here are log messages that you might see when a problem occurs during the connection stage:

LOGIN FAILED

- The *uucico* got carrier but could not log in.

FAILED (call to some system)

- The connection attempt failed as a result of a previous message.

NO CALL (RETRY TIME NOT REACHED)

- You attempted to call a system too soon after previous failed attempt.

CAN NOT CALL (SYSTEM STATUS)

- The call to a remote system aborted because of previous connection status (for example, too many failed attempts to log in to remote system, or too soon after a previous failed connection attempt). Connection status information is kept in /usr/spool/uucp/STST. There is a separate STST. file for each remote system. If the STST. file, corresponding to the remote system, is removed then a call will be permitted to that remote system (immediately).

NO CALL (MAX RECALLS)

- The local system has failed to log in to the remote system MAX times (currently 20). No further attempts can be made by *uucico* until the STST.remote file is removed from /usr/spool/uucp/STST./ .

Direct line is already in use (device name)

- *uucico* determined that there was no LCK file for this device and tried to use it (as it should) but then found that some other process (most likely tip/cu) was still using the device.

ALREADY OPEN (device name)

- This also gets the log message, direct line is already in use, except this message is logged for ACUs.

FAILED (HSM no carrier)

- The carrier was not detected when using a Hayes Smartmodem.

READ ON df02/df03 (FAILED acu=/dev/cua2, char=102)

- A dial failure occurred while using a DF02 or DF03 acu. The DF0s return a character that indicates the nature of the failure. *char=* is that character. 102 is an ASCII B that means that carrier was not detected within a prespecified amount of time (settable on the DF0s).

df02/df03 illegal return (FAILED acu=/dev/cua2, char=0)

- This message indicates that the specified acu was in a strange state, resulting in an unknown return character. This problem usually goes away when the current uucico process exits.

WRONG TIME TO CALL (systemname)

- An attempt was made to call a system at a time outside the range specified in the L.sys file.

NO DEVICE

- An attempt to call a system was made but no devices (acus or hard-wired lines) were available.

using device (device, fd=#)

- *uucico* is using *device* to call a remote system. *fd* is the file descriptor that corresponds to *device*.

LOCKED (call to systemname)

- An attempt to call a system was made for which a conversation was already in progress (for example, a lock file for that system exists in the spool directory).

FAILED (LOGIN VS MACHINE)

- A remote system tried to log in, but it failed the USERFILE security test (See Subsection 2.3.1).

TIMEOUT (DIALUP DN write)

- There was no carrier detect after dialing a number

FAILED (DIALUP ACU write)

- An error occurred while *writing* the phone number to the acu. If this error is constantly showing up it could be the result of an acu hardware failure. There might also be something wrong with the mode of the special device file.

TIMEOUT (systemname)

- A remote system has initiated a connection attempt and then stopped communication (reason unknown to the local site).

The following messages can occur at any time:

ret (#) from system!user (MAIL FAIL)

- A *mail* command failed. The exit status is indicated by *ret*. The originator is *user* on the remote node: *system*.

cmd: *command*; ret: signal #, exit # (CMD FAILED)

- A remote execution request failed. The *command* is the command that failed. The *signal* is the signal that caused the command to fail. The *exit* is the value returned by an *exit* system call in *command*.

XQT DENIED (command)

- A remote system tried to execute *command* but the request was denied by the local system.

cmd xqt'ing > 55 minutes (touch lock file)

- *uuxqt* has been executing a command for 55 minutes. The lock files associated with this command are refreshed so that they will not be removed by another *uuxqt* process.

command terminated - exceeded time limit (command)

- A command that is being processed by *uuxqt* has been running longer than 8 hours (approximately). The *command* is assumed to be a runaway process and is terminated.

system_name (execute level too low)

- The *system_name* was not allowed to execute a command because it did not have a high enough execute level. The specific command should be named by a subsequent LOGFILE entry. (See Subsection 2.3.1.3 about remote execution security).

The following messages can occur at any time:

CAUGHT (SIGNAL #)

- A signal was caught by *uucico* which caused it to terminate.

intrEXIT (signal: #)

- A signal was caught by *uucico*. This signal was probably the result of damaged software, an undetected bug, or some other system problem.

closed df0 type acu (fd=#)

- *uucico* has finished using a DF02/DF03 type of acu. *fd* is the file descriptor associated with the acu.

hasyes: closing (fd = #)

- *uucico* has finished using a Hayes Smartmodem. *fd* is the file descriptor associated with the acu.

could not close acu (errno = #, fd = #)

- *uucico* could not close the file descriptor (*fd*) associated with a DF02/DF03 type of acu. *error* is the system *errno* that was given as the reason for the failure. Consult Section 2 of the ULTRIX-32 User's Manual for the meaning of *errno*.

These messages occur during the file transfer stage after the connection has been made.

FAILED (CAN'T READ DATA)

- A work file (C.file) has specified a file to transfer but that file is not readable by *uucp* or does not exist.

FAILED (CAN'T READ SPOOLED DATA)

- A work file (C.file) has a reference to a spooled data file (D.file) and the spooled file is either unreadable by *uucp* or does not exist.

NOINPUT set no incoming files allowed (Remote name)

- /usr/lib/uucp/NOINPUT exists. The existence of this file is used as a flag to *uucico*. If NOINPUT exists do not allow incoming files.

ACCESS (DENIED)

An attempt was made by a remote system to access a file for which it does not have USERFILE permission.

REQUEST

The remote system might have run out of space. An attempt was made to write to a directory that was not writable by *uucp*, or some USERFILE restriction was violated.

DENIED (CAN'T OPEN)

A remote site tried to receive a file that the local *uucico* process can not access.

BAD READ# (expected *'char'* got *something_else*)

the local *uucico* process was waiting for a reply from the remote system but either the return message was corrupted or the remote process did not reply.

FAILED (CAN'T CREATE TM)

An attempt to create a temporary file failed, which may indicate that the system is running out of space.

FAILED (conversation complete)

A conversation with a remote system stopped before all of the spooled files were transferred. The reason for the failure is not known, but it could have been caused by something or someone killing the remote process, or possibly by a lost line. If a conversation has lasted about 90 minutes and the remote site is running an older version of *uucp*, the problem may have been the premature removal of a LCK.file at the remote site. This problem is cured in the current version.

Non error messages that occur in the LOGFILE.

Occasionally a message is placed in the LOGFILE that is not an error message. Some more error messages are:

REQUEST (char srcname dstname owner)

- A request to transfer a file to a remote site was made. If no followup message was made to indicate some kind of failure, the request was successful.

The *char* is the type of request. S for send, R for receive, X for remote execution.

The *srcname* is the name of the file on the local system.

The *dstname* is the name the file will have on the remote system.

The *owner* is the owner of the file.

REQUESTED (char srcname dstname owner)

- A remote site has requested to transfer a file to the local site. If no subsequent failure message then the transfer was successful.

OK (startup)

- The local and remote sites have successfully agreed upon what low level packet protocol to use to transfer raw data.

OK (conversation complete)

- All transactions with a remote system completed successfully.

uucp XQT (PATH.....;cmd)

- A request from a remote site to execute *cmd* was successful.

XQT QUE'D (cmd)

- A command (cmd) to be executed at a remote site was spooled.

QUE'D (S srcfile dstfile owner)

- A user request to transfer a file was spooled.

3.1.2. ERRLOG

The ERRLOG contains error messages that are less likely to occur during the normal operation of *uucp*. If a *uucp* support file such as the USERFILE is improperly set up the problem is noted here in the ERRLOG. If administrative problems such as this occur, the software will most likely stop executing, so it is important that administrative problems that are listed in the ERRLOG get cleared up quickly. When the system has run out of space preventing the creation of files or referenced files do not exist the problem will be noted here. Problems that occur during the transmission of raw data packets are also noted here. Packet problems occur infrequently and will not halt the *uucico* process. They are more indicative of a poor connection or over-loaded system and sometimes may point to a problem in the hardware.

The format of an ERRLOG message is:

ASSERT ERROR (process name) process# time_of_entry message return_code

where

ASSERT_ERROR
- is redundant information indicating that an ASSERT ERROR has occurred

process_name
- The process_name is the name of the process in which the error occurred. It could be *uucico*, *uucp*, *uuxqt*, *uux*, *uustat* or any other uucp related program.

process#
- The number is the process i.d.

time_of_entry
- The time_of_entry is the time the ASSERT error occurred.

message
- The message indicates the nature of the error.

assert_return_code
- The assert_return_code is a returned value which may be helpful for finding the source of the error (it generally is not though).

Some typical ASSERT messages are:

PKassert
- Any message that begins with PK came from the packet transmission software. Most likely due to noisy lines or a failure at the remote system.

NO default entry for remote machines
- The USERFILE does not have the default entry for remote machines.

NO default entry for local users
- The USERFILE does not have the default entry for local users.

xeq level undefined in USERFILE
- The remote execution security level (X#) was not specified for a default USERFILE entry, (See Subsection 2.3.1.3 for details on remote execution security).

CAN'T OPEN filename

- The named file probably does not exist.

WRONG ROLE

- During the file transfer stage the *uucico* process managed to reverse roles (from master to slave or vice versa) at the wrong time.

ARG COUNT<#

- Indicates that a work file (C.file) has been corrupted.

STAT FAILED filename

- A stat system call failed. If it happens continuously the named file is probably missing (and should not be).

BAD LINE

- A bad entry in the L-devices file has been encountered.

TOO MANY SUBDIRECTORIES

- The maximum number of per system subdirectories has been created. No more can be made.

TOO FEW LOG FIELDS

- The login/expect sequence of the L.sys entry for the remote system is incorrect.

BAD SPEED

- The desired line speed is not allowed.

RETURN FROM STTY

- An attempt to set the terminal I/O parameters of the outgoing line has failed. This could be either a hardware or a system problem.

BAD WRITE genbrk#

- An error occurred while generating a BREAK character on the outgoing line.

BAD DIRECTORY directory_name

- The named directory does not exist or is not set to the correct modes.

CAN NOT GET sequence file lock

- A lock file can not be created so that the sequence file (/usr/spool/uucp/sys/*sysname*/SEQF) can be accessed.

PERMISSION DENIED (Incoming C. file)

- *uucico* does not permit work files (C. files) to be transmitted to the local site because of security reasons.

3.1.3. SYSLOG

The SYSLOG records the number, size and source of each data transmission. Each entry has the following format:

uid rmte_sys (date) (int_time) sent/rec'd_data #b dur, Pk: # Rxmt #

where

uid	is the effective user id of the running process.
rmte_sys	is the name of the remote system where the data is sent or received.
date	is the date of the transaction including the time of day.
int_time	is the internal representation of the date.

sent/rec'd_data	indicates whether the data was sent or received from the remote site.
#b	is the number of bytes transferred.
dur	is the duration of the transfer in seconds.
Pk	is the number of packets needed to send the data.
Rxmt	is the number of packets that had to be retransmitted.

3.2. Guide to Debugging

When a connection to a remote system does not seem to be working, you may need to do some debugging. The amount and type of debugging may depend on whether or not the *uucp* source files are available. In any event the first place to look is the LOGFILE and ERRLOG. They may give some clue as to the nature of the problem. The system manager will have to determine if the problem is in the dialing stage, the login/handshaking stage, the file transfer stage, or whether it is a hardware problem. The most common errors occur when the remote site has set up its USERFILE incorrectly. An incorrect USERFILE results in messages sent back to local users saying that remote access to path files is denied. At any time the remote system could have changed its password or phone number. The system manager of the remote system should be contacted for updated information. When error messages constantly refer to one (out of many ACUs) the problem is probably a bad ACU. If the SYSLOG has recorded a lot of retransmissions, the hardware may be faulty, the communications line may be noisy, or the baud rate may be too high for the transmission media.

If the source files are available the local system manager should try initiating a conversation with the remote system in question. To initiate a conversation, start a *uucico* process in the MASTER mode as follows:

/usr/lib/uucp/uucico -r1 -x# -X# -s*system*

x#	is the debugging level. The higher the number the more debugging output. No packet level debugging is printed.
X#	is used to obtain packet level debugging output. The higher the number the more packet level debugging output.
system	is the system to be contacted.
r1	puts *uucico* into the master role.

Another option to *uucico* is **-f** (**f** for force). This will force *uucico* to start a conversation with a specified system regardless of the any previous connection status (as provided by the STST. files). Even if the source code is not available some useful information may be observed. The output of *uucico* follows the progress of the conversation. Debugging output from the slave *uucico* is placed in the file: AUDIT, in the spool directory, at the remote site. The output tends to be less meaningful than the LOGFILE unless the source code is available to help interpret the messages. No detailed explanation of the messages is given here since the messages are likely to change from version to version.

3.3. UUCP Self Administration Using CRONTAB Shell Scripts

The *crontab* should be modified to include shell scripts that are executed on an hourly, daily, and weekly basis. The purposes of the shell scripts are:

1. Poll remote systems for files to transfer.

Some sites do not have call units. You should poll these sites periodically to receive files from them.

2. Cleanup any temporary files.

3. Delete *uucp* transfer requests that could not be transmitted within a specified time period (currently one week).

The shell scripts are located in */usr/lib/uucp* and are discussed below:

uucp.day

This script is run at the start of the day. It cleans any lingering temporary files, saves the the previous days LOGFILE and SYSLOG in LOGFILE.yesterday and SYSLOG.yesterday, contacts any systems listed in /usr/lib/uucp/LIST.DAY, and then starts a general poll of all system for which there is work. Messages indicating the progress of this shell script are placed in /usr/lib/uucp/LOG.shells

The system manager should modify LIST.DAY so that the correct system are called. If LIST.DAY is empty or nonexistent only the general poll will be performed.

uucp.hour

This script initiates a general poll every hour. Messages indicating the start and finish of the poll are sent to the console. If any systems are listed in LIST.HOUR those systems will be contacted on an hourly basis.

uucp.noon

This script will contact any system listed in LIST.NOON at noon time.

uucp.night

uucp.night will clean up the spool directories in the early morning hours via *uuclean*. Any spool files older than 168 hours are removed. The time limit can and should be adjusted by the system manager to conform to local conditions. After the cleanup, any systems listed in LIST.NIGHT will be contacted. A general poll contacts any remaining systems for which requests are still queued. Shell script messages will be sent to LOG.shells.

uucp.week

- LOG.shells is saved in LOG.shells.week and a general poll is started.

Here are typical *crontab* entries:

```
30 * * * * su uucp < /usr/lib/uucp/uucp.hour
0 12 * * * su uucp < /usr/lib/uucp/uucp.noon
0 6 * * * su uucp < /usr/lib/uucp/uucp.day
0 1 * * 5 su uucp < /usr/lib/uucp/uucp.week
```

3.4. Monitoring the Network

The cleanup shell scripts keep a well tuned network clean of any nontransferrable or miscellaneous files. There are times when the network starts to backlog due to some of the reasons mentioned in the previous section. It is necessary therefore, for the system manager to regularly monitor the network. Two programs are useful in this regard: *uumonitor* and *uustat*.

3.4.1. UUMONITOR

/usr/lib/uucp/uumonitor is a program that creates a snapshot of the *uucp* system. The format of the output is as follows:

```
system_name  #C #X  most_recent_status  CNT:#    time
```

where

system_name
is the remote system for which the entry applies.

#C is the number of C.files queued for the remote system.

#X is the number of requests for remote execution from the remote system.

most_recent_status
is the result of the most recent attempt to connect to the remote system.

CNT:#
is the number of times that a failure to login to the remote system has occurred. This does NOT include the number of failed dial attempts.

time is the time of the last status entry was made for this system.

This command is helpful for detecting systems that have backlogs, that have gone away for awhile, have changed phone numbers, etc. The CNT: field is useful for detecting a system whose login/passwd has changed. If CNT: gets larger than the maximum allowable failures (currently 20) no further attempts to connect will be made to this system. If the number of C.files queued starts getting unusually large (depending on the system, *large* could mean anywhere from 100-1000) try to determine the cause of the backlog. If a separate subdirectory does not exist for the backlogged system, create the subdirectory should and move the spooled files from the DEFAULT directory to the per-system subdirectory. When the problem is resolved, make an entry in /usr/lib/uucp/LIST.HOUR to help clear out the backlog.

3.4.2. UUSTAT - UUCP Status Inquiry and Job Control

The *uustat* command is also useful for monitoring the network. *uustat* displays the status of, or cancels previously specified *uucp* commands, or provides general status on *uucp* connections to other systems.

NOTE: In the current implementation *uux* requests on not recorded in the *uustat* logging files. This implies that *mail* and *news* requests are not recorded by *uustat*. Some of the options are:

-**m***mch* Report the status of accessibility of machine *mch*. If *mch* is specified as **all**, then the status of all machines known to the local *uucp* are provided.

-**k***jobn* Kill the *uucp* request whose job number is *jobn*. The killed *uucp* request must belong to the person issuing the *uustat* command unless that person has superuser privilege.

-**c***hour* Remove the status entries which are older than *hour* hours. This option can only be executed by the user *uucp* or the superuser.

-**u***user* Report the status of all *uucp* requests issued by *user*.

-**s***sys* Report the status of all *uucp* requests that are destined for remote system *sys*.

-**o***hour* Report the status of all *uucp* requests that are older than *hour* hours.

-**y***hour* Report the status of all *uucp* requests which are younger than *hour* hours.

-**j***all* Report the status of **all** *uucp* requests or the specified job (request) number.

-v Report *uucp* status in English words rather than code. If this option is not specified, a status code is printed with each *uucp* request.

When no options are given, *uustat* prints the status of all *uucp* requests issued by the current user. Note that only one of the options **-j, -m, -k, -c,** can be specified along with the remaining options. For example, the command

```
uustat -usteve -slimbo -y63 -v
```

will print the status, in English, of all *uucp* jobs issued by user *steve* that were destined for system *limbo* within the last 63 hours. The format of each job status entry is:

job# user destination spool_time status_time status

The *status* may be either an octal number or a description using English words. The octal code corresponds to this description:

OCTAL	STATUS
00001	the copy failed for unknown reasons.
00002	permission to access local file is denied.
00004	permission to access remote file is denied.
00010	bad *uucp* command is generated.
00020	remote system cannot create temporary file.
00040	cannot copy to remote directory.
00100	cannot copy to local directory.
00200	local system cannot create temporary file.
00400	cannot execute *uucp*.
01000	copy succeeded.
02000	copy finished, job deleted.
04000	job is queued.

The format for the machine accessibility status entries is:

system status_time last_success_time status

where

system
is the system in question

status_time
is the time the last status entry was made.

last_success_time
is the time of the last successful connection to this system. Note that this does not imply that the entire session completed. It does mean the transfer daemons successfully completed the handshaking phase and were able to begin transferring files.

status is a self-explanatory description of the machine status.

3.4.3. Periodic Maintenance

Periodically it may be necessary to compact the spool directories.
The command: */usr/lib/uucp/uucompact* can be used to compact *uucp* spool directories. If *uucompact* is not available the following shell script outlines the procedure to pack any directory:

```
mkdir tempdir
chown uucp tempdir
mv directory/* tempdir
rm directory
mv tempdir directory
```

The **tempdir** is a temporary directory

The **directory** is the directory to be packed.

4. APPENDIX - Summary of Enhancements and Quick Installation Guide.

4.1. Enhancements

Dialers

Dialer that will work with this version are: Ventel MD212, Hayes Smartmodem, DF02/DF03, Bells 212/801, Racal Vadic 3450.

Subdirectories

/usr/spool/uucp/sys contains per system subdirectories and a DEFAULT directory. This is a huge help on busy systems.

/usr/lib/uucp/L.cmds

List of commands permitted for remote execution. A line of form 'PATH=...' sets the search path. Execution levels must be assigned to each command.

expect-send

Escape characters now permitted: \r, \n.
\r, not \n, is default char sent at end of string.
\c (put at end of string). Do not send ending \r.
\d pause 1 second (\d\d pauses 2 seconds)
"" P_ZERO 'expect nothing, start sending zero parity.'
P_EVEN (default), P_ODD, P_ONE other parity modes.
\05 Send a control-E
"" "" 'expect nothing, send a \r'.

uupoll [sysname]

Polls named system.

uumonitor

Displays spooled files, and pending *uuxqts*.

uuxqt

uuxqt can operate on a per command type basis by using -c option, for example, uuxqt -cname_of_command. It will default to a general poll of commands. Several *uuxqts* can now run concurrently.

uuclean

The -s option has been added. -sALL will clean all system subdirectories. -ssystem_name will clean spool directories belonging to system_name. *uuclean* will only mail back files if the -m option is used. Otherwise results will be mailed to *uucp*.

uucp

The -W option prevents expansion of file names that reside on remote systems. Normally files names are prepended with the current working directory if the full path of the file was not specified.

uumkspool

Creates all subdirectories for the specified system(s).

uurespool
: Move files from old spool directories to new spool format. Also useful for moving files out of DEFAULT into newly created per system spool directory.

uucompact
: Compact spool directories. -sALL or -sname_of_system. The *uucp* subsystem must be inactive for this to function properly. Single user is best. If *uucompact* is stopped it can be started again, picking up where it left off.

uucico
: The -f option has been added. -f forces *uucico* to override any previous connection status as provided by the STST. files. It is no longer necessary to remove a STST.file to ensure that a connection attempt is made.

USERFILE
: The format of the USERFILE has changed to increase security and make it more legible.

Debugging
: The -x option of the uu commands has been split out to -x and -X. -X provides packet level output, (for example, from the pk routines). -x provides all other debugging output.

This version runs on all VAXs and PDPs under ULTRIX-11 and 4.1BSD, 4.2BSD.

uucp installers should read the two documents (by Dave Nowitz) in Volume 2B of Version 7 manuals. They should then read the *UUCP Installation and Administration Guide* (this document). Understand each step below before executing. Some steps will vary slightly from system to system.

4.2. INSTALLATION

If you are starting with a new binary only ULTRIX-32 kit you only need to perform steps 6, 7, 11, and 12.

1. If you are currently running *uucp*, save the old programs:

   ```
   su root
   cd directory_where_makefile_resides
   make save
   ```

2. Editing Makefile and uucp.h - you should skip this step if you are running ULTRIX-32 and are not adding/changing source code.

   non-ULTRIX-32:

   a) Sites need to install the Berkeley directory reading library. Try (cd uucp_source_directory; cd ndir; make install). Edit Makefile to have LIBNDIR= -lndir define NDIR in uucp.h.

   b) Check LDFLAGS, OWNER, GROUP, and LIBUUCICO.

   c) Pick a method to allow *uucp* to know its system: Check out UNAME/UUNAME/WHOAMI/CCWHOAMI in uucp.h

d) Define SYSIII if appropriate in uucp.h.

e) Your *make* may fail because the Makefile is so large. If so, in /usr/src/cmds/make/defs, change

3. Make the new commands: (Binary only sites should skip this step.)

   make

4. WAIT UNTIL THE UUCP SYSTEM IS IDLE!! Single-user is best.

   su root (it is important that chmod and chown work below)

5. Install the new commands: (Binary only sites skip this step.)

   make install

6. Edit and install the control files:

   Edit and install into /usr/lib/uucp if necessary USERFILE, L.cmds, L.sys, L-devices, L-dialcodes.

   NOTE: These files must be owned by the same owner and group as the *uucp* commands and *uucp* spool files. The format for dialers is slightly different so that any dialer can be handled.

7. Make new subdirectories:

   For safety: cd /usr/spool/uucp; tar c . (Save queued files on tape.) The following assumes your site name is produced by 'uuname -l'.

   cd directory_that_contains_Makefile (/usr/lib/uucp on binary kits)

   make mkdirs

   This will make all spool directories including the DEFAULT system spool directory: /usr/spool/uucp/sys/DEFAULT.

   For each additional site that will have its own subdirectory :

   /usr/lib/uucp/uumkspool system

8. Move old queued files:

   If you have spooled files, they must be moved into the appropriate subdirectories. Assuming all spool files are in /usr/spool/uucp, (for example, you did not have subdirectories before), the following command will move the spool files to the new system directories:

   /usr/lib/uucp/uurespool -t1

   For each system directory that was created with uumkspool the following actions occur (assuming the name of your system is duke):

Files beginning C. are put in the C. subdirectory.
Files beginning with D.dukeX are put in that directory, not D.
Files beginning with D.duke are put in the D.duke directory.
All other D.files are put in the D. directory.
X.files are put in the X. directory.

Delete other old directories if you had any (for example, LOG).

9. Compact spool directories:

From time to time it may be necessary to compact spool directories. The following command facilitates this:

```
/usr/lib/uucp/uucompact -sALL or -sname_of_system
```

NOTE: The *uucp* subsystem should be quiescent before and during the execution of *uucompact*.

10. Clear out status files:

```
cd /usr/lib/uucp
cp /dev/null L_stat
cp /dev/null R_stat
```

11. Test the new system.

Test by mailing a letter somewhere and back (This assumes you have mail working properly of course). If it works, the new system is probably fine. Otherwise, figure out what is wrong. Start by examining LOGFILE. Try /usr/lib/uucp/uucico -r1 -sname -x7 If things are no-go, you can back out the changes by restoring the old uu programs and the spooled files.

12. Install administrative scripts (uucp.hour, uucp.day, ...) as described in this manual. These scripts should be run using *cron*.

AA-CN32A-TE